CS388: Natural Language Processing

Lecture 6: NN Implementation
Announcements

- A1 due today at midnight
- A2 out at midnight
Recall: Feedforward NNs

\[ P(y|x) = \text{softmax}(Wg(Vf(x))) \]
Recall: Training Feedforward NNs

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

- Maximize log likelihood of training data. For one point:

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

- How to compute the gradient with respect to \( W \) and \( V \)?
Recall: Backpropagation

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

- Neural net implementation / PyTorch 101
- Neural net training tips
- Deep averaging networks
Implementing Neural Networks: PyTorch 101
Computing gradients is hard!

Automatic differentiation: instrument code to keep track of derivatives

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

Computation is now something we need to reason about symbolically; use a library like PyTorch (or Tensorflow)

Ensuing code examples are on the course website: ffnn_example.py under “Readings”
PyTorch

- Framework for defining computations that provides easy access to derivatives

- Module: defines a neural network (can use wrap other modules which implement predefined layers)

- If forward() uses crazy stuff, you have to write backward yourself

```python
torch.nn.Module
    # Takes an example x and computes result
    forward(x):
        ...
    # Computes gradient after forward() is called
    backward(): # produced automatically
        ...
```
Define forward pass for $P(y|x) = \text{softmax}(Wg(Vf(x)))$

class FFNN(nn.Module):
    def __init__(self, input_size, hidden_size, out_size):
        super(FFNN, self).__init__()
        self.V = nn.Linear(input_size, hidden_size)
        self.g = nn.Tanh()  # or nn.ReLU(), sigmoid()...
        self.W = nn.Linear(hidden_size, out_size)
        self.softmax = nn.Softmax(dim=0)

    def forward(self, x):
        return self.softmax(self.W(self.g(self.V(x))))
        (syntactic sugar for forward)
Whatever you define with torch.nn needs its input as some sort of tensor, whether it’s integer word indices or real-valued vectors

```python
def form_input(x) -> torch.Tensor:
    # Index words/word embeddings/etc.
    return torch.from_numpy(x).float()
```

torch.Tensor is a different data structure from a numpy array, but you can translate back and forth fairly easily.

Note that *translating out of PyTorch will break backpropagation*; don’t do this inside your Module.
Training and Optimization

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

\[
\text{ffnn} = \text{FFNN(inp, hid, out)} \\
\text{optimizer} = \text{optim.Adam(ffnn.parameters(), lr=lr)} \\
\text{for epoch in range(0, num\_epochs):} \\
\quad \text{for (input, gold\_label) in training\_data:} \\
\quad \quad \text{ffnn.zero\_grad()} \ # \text{clear gradient variables} \\
\quad \quad \text{probs} = \text{ffnn.forward(input)} \\
\quad \quad \text{loss} = \text{torch.neg(torch.log(probs)).dot(gold\_label)} \\
\quad \quad \text{loss.backward()} \\
\quad \quad \text{optimizer.step()} \\
\]
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)
        nn.init.uniform(self.V.weight)

- Initializing to a nonzero value is critical, more in a bit
Training a Model

Define modules, etc.
Initialize weights and optimizer

For each epoch:

  For each batch of data:

    Zero out gradient
    Compute loss on batch
    Autograd to compute gradients and take step on optimizer

  [Optional: check performance on dev set to identify overfitting]

Run on dev/test set
Batching and Optimization
(blackboard)
Batching

- Batching: processing multiple examples in parallel (for training or test), 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 range from 1-64 or so (depending on GPU memory, etc.)
Optimization Takeaways

- Need to initialize to values that aren’t 0 but aren’t too large
  - Can do random uniform / normal initialization with appropriate scale; also fancier initializers (Xavier Glorot initializer, Kaiming He) to match variances across layers
- Use Adam as your optimizer
- Consider adding dropout layers (at input or hidden layers; never at output). Typically 0.2 - 0.5 are good ranges for dropout probability
DANs
Word Embeddings

Currently we think of words as “one-hot” vectors

\[\text{the} = [1, 0, 0, 0, 0, 0, ...]\]
\[\text{good} = [0, 0, 0, 1, 0, 0, ...]\]
\[\text{great} = [0, 0, 0, 0, 0, 1, ...]\]

\[\text{good} \text{ and } \text{great} \text{ seem as dissimilar as } \text{good} \text{ and } \text{the}\]

Neural networks are built to learn sophisticated nonlinear functions of continuous inputs; our inputs are weird and discrete
Word Embeddings

- Want a vector space where similar words have similar embeddings
  
  \[ \text{great} \approx \text{good} \]

- Next lecture: come up with a way to produce these embeddings

- For each word, want “medium” dimensional vector (50-300 dims) representing it

\[
\begin{align*}
great & \quad \text{good} \\
\text{enjoyable} & \quad \text{is} \\
dog & \\
\text{bad} &
\end{align*}
\]
Deep Averaging Networks: feedforward neural network on average of word embeddings from input

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

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

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

Predator  is  a  masterpiece
\[ c_1 \quad c_2 \quad c_3 \quad c_4 \]

Iyyer et al. (2015)
Widely-held view: need to model syntactic structure to represent language

Surprising that averaging can work as well as this sort of composition

Deep Averaging Networks

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>NBOY-RAND</td>
<td>76.2</td>
<td>42.3</td>
<td>81.4</td>
<td>88.9</td>
<td>91</td>
</tr>
<tr>
<td>NBOY</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>WRRBDM*</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>89.2</td>
<td>—</td>
</tr>
</tbody>
</table>

- **No pretrained embeddings**
- **Bag-of-words**
- **Tree-structured neural networks**

Iyyer et al. (2015)
Wang and Manning (2012)
Kim (2014)
# Deep Averaging Networks

Iyyer et al. (2015)

Will return to compositionality with syntax and LSTMs

<table>
<thead>
<tr>
<th>Sentence</th>
<th>DAN</th>
<th>DRecNN</th>
<th>Ground Truth</th>
</tr>
</thead>
<tbody>
<tr>
<td>who knows what exactly godard is on about in this film, but his words and images don’t have to add up to mesmerize you.</td>
<td>positive</td>
<td>positive</td>
<td>positive</td>
</tr>
<tr>
<td>it’s so good that its relentless, polished wit can withstand not only inept school productions, but even oliver parker’s movie adaptation too bad, but thanks to some lovely comedic moments and several fine performances, it’s not a total loss</td>
<td>negative</td>
<td>positive</td>
<td>positive</td>
</tr>
<tr>
<td>this movie was not good</td>
<td>negative</td>
<td>negative</td>
<td>negative</td>
</tr>
<tr>
<td>this movie was good</td>
<td>positive</td>
<td>positive</td>
<td>positive</td>
</tr>
<tr>
<td>this movie was bad</td>
<td>negative</td>
<td>negative</td>
<td>negative</td>
</tr>
<tr>
<td>the movie was not bad</td>
<td>negative</td>
<td>negative</td>
<td>positive</td>
</tr>
</tbody>
</table>

† Will return to compositionality with syntax and LSTMs

Iyyer et al. (2015)
torch.nn.Embedding: maps vector of indices to matrix of word vectors

Predator is a masterpiece

1820 24 1 2047

\[ \begin{array}{cccc}
\text{Predator} & \text{is} & \text{a} & \text{masterpiece} \\
1820 & 24 & 1 & 2047
\end{array} \]

- \( n \) indices => \( n \times d \) matrix of \( d \)-dimensional word embeddings
- \( b \times n \) indices => \( b \times n \times d \) tensor of \( d \)-dimensional word embeddings
Next Time

- Guest lecture
- Next Thursday: word embeddings