CS378: Natural Language Processing

Lecture 6: NN Implementation
Announcements

- Assignment 1 due today
- Assignment 2 out today, due in two weeks
- Fairness response due Thursday (submit on Canvas)
- Seating chart
Recap
Classification Review

- See Instapoll
Feedforward Networks
Vectorization and Softmax

\[
P(y|x) = \frac{\exp(w_y^T f(x))}{\sum_{y' \in Y} \exp(w_{y'}^T f(x))}
\]

- Single scalar probability

- Three classes, "different weights"
  \[
  \begin{align*}
  w_1^T f(x) &= -1.1 \\
  w_2^T f(x) &= 2.1 \\
  w_3^T f(x) &= -0.4
  \end{align*}
  \]

- Softmax operation = "exponentiate and normalize"

- We write this as: \( \text{softmax}(W f(x)) \)
Logistic Regression as a Neural Net

\[ P(y|x) = \frac{\exp(w_y^T f(x))}{\sum_{y' \in Y} \exp(w_{y'}^T f(x))} \]  

- Single scalar probability

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

- Weight vector per class;  
  \( W \) is [num classes x num feats]

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

- Now one hidden layer
Neural Networks for Classification

\[ P(y|x) = \text{softmax}(Wg(Vf(x))) \]
Backpropagation
(with pictures! Full derivations at the end of the slides)
Training Objective

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

- Consider the log likelihood of a single training example:

\[ \mathcal{L}(x, i^*) = \log P(y = i^*|x) \]

  where \( i^* \) is the index of the gold label for an example

- Backpropagation is an algorithm for computing gradients of \( W \) and \( V \) (and in general any network parameters)
Backpropagation: Picture

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

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

- **n features**
- **d hidden units**
- **num_classes**
- **probs**

- Gradient w.r.t. \( W \): looks like logistic regression, can be computed treating \( z \) as the features
\[ P(y|x) = \text{softmax}(Wg(Vf(x))) \]

- Can forget everything after \( z \), treat it as the output and keep backpropping
$P(y|x) = \text{softmax}(Wg(Vf(x)))$

- $n$ features
- $d$ hidden units
- $\text{num\_classes}$
- probs

- Combine backward gradients with forward-pass products
Pytorch Basics

(code examples are on the course website: ffnn_example.py )
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 \( P(y|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x}))) \)

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/embed words/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

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

```python
ffnn = FFNN(inp, hid, out)
optimizer = optim.Adam(ffnn.parameters(), lr=lr)
for epoch in range(0, num_epochs):
    for (input, gold_label) in training_data:
        ffnn.zero_grad() # clear gradient variables
        probs = ffnn.forward(input)
        loss = torch.neg(torch.log(probs)).dot(gold_label)
        loss.backward()
        optimizer.step()
```

one-hot vector of the label (e.g., [0, 1, 0])
negative log-likelihood of correct answer
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. See optimization video on course website
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
Pytorch example
Batching
Batching

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)

- Can run this in a batched fashion without modification!
Batching

- Modify the training loop to run over multiple examples at once

```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
DANs
Word Embeddings

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

the = [1, 0, 0, 0, 0, 0, ...]
good = [0, 0, 0, 1, 0, 0, ...]
great = [0, 0, 0, 0, 0, 1, ...]

- good and great seem as dissimilar as good and 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
  
  $great \approx good$

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

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

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

\[
\text{softmax} \\
h_2 = f(W_2 \cdot h_1 + b_2) \\
h_1 = f(W_1 \cdot av + b_1) \\
\text{av} = \frac{1}{4} \sum_{i=1}^{4} c_i
\]

Predator \quad is \quad a \quad masterpiece

\( c_1 \quad c_2 \quad c_3 \quad c_4 \)

Iyyer et al. (2015)
Deep Averaging Networks

- Widely-held view: need to model syntactic structure to represent language

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

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>NBOV-RAND</td>
<td>76.2</td>
<td>42.3</td>
<td>81.4</td>
<td>88.9</td>
<td>91</td>
</tr>
<tr>
<td>NBOV</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>—</td>
<td>50.6</td>
<td>86.9</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>—</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>WRRBMM*</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**

Wang and Manning (2012)
Iyyer et al. (2015)
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>
torch.nn.Embedding: maps vector of indices to matrix of word vectors

Predator is a masterpiece

1820  24  1  2047

\[ \begin{bmatrix}
\text{Predator} \\
\text{is} \\
\text{a} \\
\text{masterpiece}
\end{bmatrix} \]

\( n \) indices \( \Rightarrow n \times d \) matrix of \( d \)-dimensional word embeddings

\( b \times n \) indices \( \Rightarrow b \times n \times d \) tensor of \( d \)-dimensional word embeddings
Word Embeddings
Neural Nets History
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”)
- 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. (2014): seq2seq for neural MT (LSTMs work for NLP?)

- Chen and Manning transition-based dependency parser (feedforward)

- 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
Backpropagation — Derivations
(not covered in lecture, optional)
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

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

- **Gradient with respect to** \( W \):

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

- Looks like logistic regression with \( \mathbf{z} \) as the features!
Computing Gradients: Backpropagation

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

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

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

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

\[ \text{dim} = d \]

\[ z = g(Vf(x)) \]

Activations at hidden layer

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

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

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

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

**Activations at hidden layer**

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

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

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

- Second term: gradient of linear function

- First term: \( \text{err}(\mathbf{z}) \); represents gradient w.r.t. \( \mathbf{z} \)
Backpropagation

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

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