CS395T: Structured Models for NLP
Lecture 14: Neural Network Implementation

Greg Durrett
Administrivia

- Project 2 due today
- Project 3 out today
  - Sentiment analysis using feedforward neural networks plus your choice of RNNs or CNNs
  - Project zip contains sample Tensorflow code demonstrating what’s in today’s lecture
Recall: Feedforward NNs

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

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

- Implementation details
- Training
- Word representations
Implementation Details
Computation Graphs

- Computing gradients is hard!

- Automatic differentiation: instrument code to keep track of derivatives

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

- In practice: need other operations, want more control -> use an external computation graph library
Define computation abstractly, in terms of symbols

Can compute gradients of c with respect to (x, y, z) easily

Useful abstraction: supports both CPU and GPU implementations

Disadvantage: higher-level specification, so hard to control memory allocation and low-level implementation details
x = tf.placeholder("x")
y = tf.placeholder("y")
z = tf.placeholder("z")
a = tf.add(x, y)
b = tf.multiply(a, z)
c = tf.add(b, a)

with tf.Session() as sess:
    output = sess.run([a, c],
                      dict_of_input_values)
\[ P(y|x) = \text{softmax}(Wg(Vf(x))) \]

```python
fx = tf.placeholder(tf.float32, feat_vec_size)
V = tf.get_variable("V", [hidden_size, feat_vec_size])
z = tf.sigmoid(tf.tensordot(V, fx, 1))
W = tf.get_variable("W", [num_classes, hidden_size])
probs = tf.nn.softmax(tf.tensordot(W, z, 1))
```

- placeholder: input to the system; variable: parameter to learn
Computation Graph: FFNN

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

```python
fx = tf.placeholder(tf.float32, feat_vec_size)
V = tf.get_variable("V", [hidden_size, feat_vec_size])
z = tf.sigmoid(tf.tensordot(V, fx, 1))
W = tf.get_variable("W", [num_classes, hidden_size])
probs = tf.nn.softmax(tf.tensordot(W, z, 1))
label = tf.placeholder(tf.int32, num_classes)
loss = tf.negative(tf.log(tf.tensordot(probs, label, 1)))
```

- Tensorflow can compute gradients for \(W\) and \(V\) based on loss
- Shortcut helper methods exist like `tf.nn.softmax_cross_entropy_with_logits`
Training a Model

Define a computation graph

Define an operator that updates the parameters based on an example

For each epoch:
  
  For each example:
    
    Evaluate the training operator on the example

Decode test set
Batching data gives speedups due to more efficient matrix operations, leads to better learning outcomes too

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

```python
fx = tf.placeholder(tf.float32, [batch_size, feat_vec_size])
V = tf.get_variable("V", [hidden_size, feat_vec_size])
z = tf.sigmoid(tf.tensordot(V, fx, [1,1])) # batch_size x hidden_size
... 
loss = [sum over losses from batch]
```
Batch Training a Model

Define a computation graph to process a **batch of data**

Define an operator that updates the parameters based on a **batch**

For each epoch:

   For each **batch**:

        Evaluate the training operator on the **batch**

Decode test set in **batches**
Training Tips
Training Basics

- Basic formula: compute gradients on batch, use first-order opt. method

- How to initialize? How to regularize? What optimizer to use?

- This lecture: some practical tricks. Take deep learning or optimization courses to understand this further
How does initialization affect learning?

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

- How do we initialize \( V \) and \( W \)?
- What consequences does this have?
How does initialization affect learning?

- Why is it important to have small activations?

- If cell activations are too large in absolute value, gradients are small too.

- ReLU: larger dynamic range (all positive numbers), but can produce big values, can break down if everything is too negative.
Initialization

1) Can’t use zeroes for parameters to produce hidden layers: all values in that hidden layer are always 0 and have gradients of 0, never change

2) Initialize too large and cells are saturated

- Can do random uniform / normal initialization with appropriate scale
- Glorot initializer: $U \left[ -\sqrt{\frac{6}{\text{fan-in} + \text{fan-out}}}, +\sqrt{\frac{6}{\text{fan-in} + \text{fan-out}}} \right]$
  - Want variance of inputs and gradients for each layer to be the same
- Batch normalization (Ioffe and Szegedy, 2015): periodically shift+rescale each layer to have mean 0 and variance 1 over a batch (useful if net is deep)
Dropout

- Probabilistically zero out parts of the network during training to prevent overfitting, use whole network at test time

- Form of stochastic regularization

- Similar to benefits of ensembling: network needs to be robust to missing signals, so it has redundancy

Srivastava et al. (2014)
Dropout

- In tensorflow: implemented as an additional layer in a network

```python
hidden_dropped_out = tf.nn.dropout(hidden, dropout_keep_prob)
```

- Often use low dropout (keep a value with probability 0.8) at the input and moderate dropout (keep with probability 0.5) internally in feedforward networks (not in RNNs)
Adam (Kingma and Ba, ICLR 2015) is very widely used
Adaptive step size like Adagrad, incorporates momentum
Wilson et al. NIPS 2017: adaptive methods can actually perform badly at test time (Adam is in pink, SGD in black)

Check dev set periodically, decrease learning rate if not making progress
Visualization with Tensorboard

- Visualize the computation graph and logs of the objective over time
Four elements of a structured machine learning method:

- Model: feedforward, RNNs, CNNs can be defined in a uniform framework
- Objective: many loss functions look similar, just changes the last layer of the neural network
- Inference: define the network, Tensorflow takes care of it (mostly...)
- Training: lots of choices for optimization/hyperparameters
Word Representations
Word Representations

- Neural networks work very well at continuous data, but words are discrete.
- Continuous model $\text{->}$ expects continuous semantics from input.
Word Embeddings

- Part-of-speech tagging with FFNNs
  
  Fed raises **interest** rates in order to ...  
  
- Word embeddings for each word form input

Botha et al. (2017)
Want a vector space where similar words have similar embeddings

- the movie was great
- the movie was good
Word Representations

- Neural networks work very well at continuous data, but words are discrete.
- Continuous model <-> expects continuous semantics from input.
- “Can tell a word by the company it keeps” Firth 1957.

[Finch and Chater 92, Shuetze 93, many others]
Continuous Bag-of-Words

- Predict word from context

\[ \text{the:dog bit the:man} \]

Mikolov et al. (2013)

- Parameters: \( d \times |V| \) vectors, \( |V| \times d \) output parameters (W)
- Maximize likelihood of gold labels (no manual labeling required!)

\[ P(w|w_{-1}, w_{+1}) \]

\[ \text{gold} = \text{bit} \]
Skip-Gram

- Predict one word of context from word

\[ \text{the dog bit the man} \]

- Parameters: \( d \times |V| \) vectors, \( |V| \times d \) output parameters (W)

Another training example: \( \text{bit} \rightarrow \text{the} \)

Mikolov et al. (2013)
Skip-Gram with Negative Sampling

- Problem: want to train on 1B+ words, multiplying by $|V| \times d$ matrix for each is too expensive

- Solution: take (word, context) pairs and classify them as “real” or not. Create random negative examples by sampling

$$(bit, the) \Rightarrow +1 \quad (bit, a) \Rightarrow -1$$

$$(bit, dog) \Rightarrow +1 \quad (bit, fish) \Rightarrow -1$$

$$P(pos | w, c) = \frac{e^{w \cdot c}}{e^{w \cdot c} + 1}$$

- $d \times |V|$ vectors, $d \times |V|$ context vectors (same # of params as before)
(king - man) + woman = queen

king + (woman - man) = queen

- Why would this be?

- woman - man captures the difference in the contexts that these occur in

- Dominant change: more “he” with man and “she” with woman — similar to difference between king and queen
GloVe

- Word co-occurrences are what matter directly

<table>
<thead>
<tr>
<th>Probability and Ratio</th>
<th>$k = \text{solid}$</th>
<th>$k = \text{gas}$</th>
<th>$k = \text{water}$</th>
<th>$k = \text{fashion}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P(k</td>
<td>\text{ice})$</td>
<td>$1.9 \times 10^{-4}$</td>
<td>$6.6 \times 10^{-5}$</td>
<td>$3.0 \times 10^{-3}$</td>
</tr>
<tr>
<td>$P(k</td>
<td>\text{steam})$</td>
<td>$2.2 \times 10^{-5}$</td>
<td>$7.8 \times 10^{-4}$</td>
<td>$2.2 \times 10^{-3}$</td>
</tr>
<tr>
<td>$P(k</td>
<td>\text{ice})/P(k</td>
<td>\text{steam})$</td>
<td>8.9</td>
<td>$8.5 \times 10^{-2}$</td>
</tr>
</tbody>
</table>

- Weighted least-squares problem to directly predict word co-occurrence matrix (like matrix factorization)

Pennington et al. (2014)
Using Word Embeddings

- Approach 1: learn embeddings as parameters from your data
- Approach 2: initialize using GloVe/CBOW/SGNS, keep fixed
  - Faster because no need to update these parameters
- Approach 3: initialize using GloVe/CBOW/SGNS, fine-tune
  - Typically works best

```python
# Indexed sentence of length sent_len, e.g.: [12, 36, 47, 8]
input_words = tf.placeholder(tf.int32, [sent_len])
encoder = tf.get_variable("embed", [voc_size, embedding_size])
embedded_input_words = tf.nn.embedding_lookup(encoder, input_words)
# embedded_input_words: sent_len x embedding_size tensor
```
Takeaways

- Lots to tune with neural networks
  - Training: optimizer, initializer, regularization (dropout), ...
  - Hyperparameters: dimensionality of word embeddings, layers, ...
- Word vectors: various choices of pre-trained vectors work well as initializers
- Next time: RNNs / LSTMs / GRUs