Class 3: Training Recurrent Nets

Arvind Ramanathan
Computational Science & Engineering, Oak Ridge National Laboratory, Oak Ridge, TN 37830
ramanathana@ornl.gov
Last class

Basics of RNNs

Recurrent network modeling

How to build a RNN and its different types
Quick Recap (1): Vanilla (E.g., Convolutional) nets

- Most convolutional nets are limited in their ability to represent data:
  - Take a fixed size input vector and output a fixed size vector
    - E.g., take image and classify
  - Only fixed number of layers/computational steps
    - E.g., LeNet has five layers
- Efficient to train -- but representation is still limited to neighborhood information
  - Does not capture potentially long range interactions
- Usually applicable in “discriminative” situations...
  - Referred to as “one-to-one” architectures
Quick Recap (2): RNN and its components

RNNs combine the input vector with their state vector with a fixed (but learned) function to produce a new state vector.

Think of running a “fixed” program + some internal variables on every input.

RNNs represent programs: RNNs are Turing complete -- meaning they can run any arbitrary program!
Quick Recap (3): RNN + recurrence formula

\[ h_t = f_W(h_{t-1}, x_t) \]

- We can process a sequence of vectors \( x \) by applying a recurrence formula at every time step.
- The same function and same set of parameters are used every time step.
A simple RNN

The state consists of a single hidden vector $h$:

$$h_t = f_W(h_{t-1}, x_t)$$

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$
Advancing / Unrolling the RNN → Computational Graph Representation
Example: Character level language model

Vocabulary: [h, e, l, o]

Example training sequence:

“hello”

\[
h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)
\]
Example:
Character level language model

Vocabulary: [h, e, l, o]

Example training sequence:
“hello”
Example: Character level language model sampling

Vocabulary: [h, e, l, o]

At test-time sample characters one at a time, feed back to model
Training your first RNN...
Let's take a simple example and explore...

\[ s_t = \tanh(Ux_t + Ws_{t-1}) \]
\[ \hat{o}_t = \text{softmax}(Vs_t) \]
Expanding log loss of the model...

\[
E_t(o_t, \hat{o}_t) = -o_t \log \hat{o}_t
\]

\[
E(o, \hat{o}) = \sum_t E_t(o_t, \hat{o}_t)
\]

\[
= - \sum_t o_t \log \hat{o}_t
\]

How do we compute the gradients?

We need to compute gradients of the error with respect to our parameters $U$, $V$, $W$

Use Stochastic Gradient Descent

sum up the gradients at each time step for one training example

$$\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$$
Computing gradients at E3

\[ \frac{\partial E_3}{\partial V} = \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial V} = \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial z_3} \frac{\partial z_3}{\partial V_3} = (\hat{y}_3 - y_3) \otimes s_3 \]

Important note: Gradient values at E3 depend only on the current timestep...

Computing gradient wrt V is easy.....
What about computing gradient wrt \( W \)?

\[
\frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial W}
\]

\( s_3 = \tanh(Ux_t + Ws_2) \)

\( s_2 = \tanh(Ux_t + Ws_1) \) ...

\[
\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial W}
\]
Unrolling the gradients through the computational graph

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial W}$$

Exactly the same backpropagation algorithm -- key difference is that for W at each time step we sum up the gradients until that step
How do we write it in Python?

A naive implementation

Includes two for loops

- One for time-range (sequence length)
- One for propagating the gradients

This should give you a sense of why BPTT is expensive computationally

- A serial computation embedded within what could be potentially parallel

Arbitrary length sequences can make it even more expensive to compute backprop...

def bptt(self, x, y):
    T = len(y)
    # Perform forward propagation
    o, s = self.forward_propagation(x)
    # We accumulate the gradients in these variables
    dLdU = np.zeros(self.U.shape)
    dLdV = np.zeros(self.V.shape)
    dLdW = np.zeros(self.W.shape)
    delta_o = o
    delta_o[np.arange(len(y)), y] -= 1.
    # For each output backwards...
    for t in np.arange(T)[::-1]:
        dLdV += np.outer(delta_o[t], s[t].T)
        # Initial delta calculation: dL/dz
        delta_t = self.V.T.dot(delta_o[t]) * (1 - (s[t] ** 2))
        # Backpropagation through time (for at most self.bptt_truncate steps)
        for bptt_step in np.arange(max(0, t-self.bptt_truncate), t+1)[::-1]:
            # print "Backpropagation step t=%d bptt step=%d " % (t, bptt_step)
            # Add to gradients at each previous step
            dLdW += np.outer(delta_t, s[bptt_step-1])
            dLdU[:,x[bptt_step]] += delta_t
            # Update delta for next step dL/dz at t-1
            delta_t = self.W.T.dot(delta_t) * (1 - s[bptt_step-1] ** 2)
    return [dLdU, dLdV, dLdW]
Problems galore with BPTT...

There is a product of gradients that propagates ...

\[
\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_k}{\partial \hat{s}_3} \frac{\partial \hat{s}_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial \hat{s}_3} \frac{\partial \hat{s}_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_k} \frac{\partial s_k}{\partial W}
\]

\[
\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \left( \prod_{j=k+1}^{3} \frac{\partial s_j}{\partial \hat{s}_j} \right) \frac{\partial \hat{s}_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial \hat{s}_3} \frac{\partial \hat{s}_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_k} \frac{\partial s_k}{\partial W}
\]
Your first tryst with the Vanishing Gradient...

Output $a_j$ from the $j^{th}$ neuron is $\sigma(z_j)$. Input is the weighted neurons

$$z_j = w_j a_{j-1} + b_j$$

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$
Why does vanishing gradient occur

\[
\frac{\partial C}{\partial b_1} = \sigma'(z_1) w_2 \sigma'(z_2) \left( w_3 \sigma'(z_3) w_4 \sigma'(z_4) \frac{\partial C}{\partial a_4} \right) < \frac{1}{4}
\]

\[
\frac{\partial C}{\partial b_3} = \sigma'(z_3) w_4 \sigma'(z_4) \frac{\partial C}{\partial a_4}
\]

A similar argument holds for “exploding” gradients
Let's take a relatively complex example...

- Maps an input sequence of $x$ values to a corresponding sequence of output $o$ values.
- A loss $L$ measures how far each $o$ is from the corresponding training target $y$.
- The loss $L$ internally computes $y = \text{softmax}(o)$ and compares this to the target $y$.
- Input to hidden connections parametrized by a weight matrix $U$.
- Hidden-to-hidden recurrent connections parametrized by a weight matrix $W$.
- Hidden-to-output connections parameterize by a weight matrix.
Forward Propagation

\[
\tilde{a}(t) = \tilde{b} + WH^{(t-1)} + UX^{(t)}
\]

\[
\tilde{h}(t) = \tanh(\tilde{a}(t))
\]

\[
\tilde{o}(t) = \tilde{c} + VH^{(t)}
\]

\[
\hat{y} = \text{softmax}(\tilde{o}(t))
\]
What is the total loss for the output sequence?

\[ L(\{\overline{x}^{(1)}, \ldots, \overline{x}^{(\tau)}\}, \{\overline{y}^{(1)}, \ldots, \overline{y}^{(\tau)}\}) = \sum_t L^{(t)} \]

\[ = -\sum_t \log p_{model}(y^{(t)}|\{\overline{x}^{(1)}, \ldots, \overline{x}^{(\tau)}\}) \]

- Recall that training requires us to compute the gradients over this log likelihood (loss) function
- Expensive!!
  - Forward propagation from left to right of the unrolled graph
  - Backward propagation from right to left
  - O(\tau) computation is inherently serial; cannot be parallel, needs O(\tau) memory too
- New training algorithm: Backward propagation through time (BPTT)
- Same holds for recurrence between hidden units
Understanding the computational graph...

\[ \vec{a}^{(t)} = \vec{b} + \mathbf{W}\tilde{h}^{(t-1)} + \mathbf{U}\tilde{x}^{(t)} \]

\[ \vec{h}^{(t)} = \tanh(\vec{a}^{(t)}) \]

\[ \vec{o}^{(t)} = \vec{c} + \mathbf{V}\tilde{h}^{(t)} \]

\[ \hat{y} = \text{softmax}(\vec{o}^{(t)}) \]
Computing the gradients (1)

For each node $N$, we need to evaluate gradient...

The gradient $\nabla_{\bar{o}(t)} L$ for all $(i, t)$, is as follows

We start working backward from the end of the sequence. At the final step $h$ only has $o$ as its descendent.

$$\nabla_N L = \frac{\partial L}{\partial L(t)} = 1$$

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

$$\nabla_{\bar{h}(\tau)} = V^T \nabla_{\bar{o}(\tau)} L$$
Computing the gradients (2)

iterate backward in time to back-propagate gradients through time

\[ \nabla_{\tilde{h}(t)} L = \left( \frac{\partial \tilde{h}(t+1)}{\partial \tilde{h}(t)} \right)^T \left( \nabla_{\tilde{h}(t+1)} L \right) + \left( \frac{\partial \sigma(t)}{\partial \tilde{h}(t)} \right)^T \left( \nabla_{\sigma(t)} L \right) \]

\[ W^T (\nabla_{\tilde{h}(t+1)} L) \operatorname{diag} \left( 1 - (\tilde{h}(t+1))^2 \right) + V^T (\nabla_{\sigma(t)} L) \]

diagonal matrix calculating the gradients along the elements of the hidden unit
Computing gradients is hard...

At any given time $t$, there is a need to look $\tau$ steps behind to get the right gradients

The $\tau$ steps to be taken can be arbitrarily large:

- We may want to capture dependencies in the sequence long enough
- How long these dependencies are is unknown a priori

Training a RNN can be hard: need practical solutions to solve this problem

- Try to stop BPTT to some number of steps
- Change the internal network representation to ensure “gated” information flow
Solution 1: Truncate Backprop...

- Run forward and backward through chunks of the sequence instead of whole sequence
- Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps
Solution 2: Handling vanishing/exploding gradients by changing recurrent functions

The tanh () function has a gradient behavior that can potentially vanish/explode.

Replace the single tanh with additional layers.

Long Short Term Memory (LSTM)

Gated Recurrent Units (GRU)

\[
s_t = \tanh(U x_t + W s_{t-1}) \\
\hat{o}_t = \text{softmax}(V s_t)
\]
“Gating” Information

LSTM: Long Short Term Memory

GRU: Gated Recurrent Units
Long Short Term Memory (LSTM)

http://colah.github.io/posts/2015-08-Understanding-LSTMs/
LSTM (1): Controlling information let through

Intuitively, forget gate keeps track of what information to “lose”
Or how to weigh the information such that they can be propagated further

\[ f_t = \sigma \left( W_f \cdot [h_{t-1}, x_t] + b_f \right) \]
Next step is to keep track of what information we are going to store in the cell. Sigmoid layer determines which values to update. Tanh creates a vector of new candidate values.
LSTM (3): Controlling information let through

Next step: update the old cell state with the new cell state
Ct-1 is already available, just a simple vector add is sufficient to get this state

\[ C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \]
LSTM (4): Controlling information let through

Decide what we are going to output: determined by a filter sigmoid layer which decides what parts of the cell state we’re going to output

\[
o_t = \sigma \left( W_o \left[ h_{t-1}, x_t \right] + b_o \right)
\]

\[
h_t = o_t \times \tanh \left( C_t \right)
\]

Tanh decides what values should be output (by quashing values between -1 and +1)

http://colah.github.io/posts/2015-08-Understanding-LSTMs/
Variants of LSTM

\[ f_t = \sigma (W_f \cdot [C_{t-1}, h_{t-1}, x_t] + b_f) \]
\[ i_t = \sigma (W_i \cdot [C_{t-1}, h_{t-1}, x_t] + b_i) \]
\[ o_t = \sigma (W_o \cdot [C_t, h_{t-1}, x_t] + b_o) \]

http://colah.github.io/posts/2015-08-Understanding-LSTMs/
Variants of LSTM (2)

\[ C_t = f_t \times C_{t-1} + (1-f_t) \times \tilde{C}_t \]

http://colah.github.io/posts/2015-08-Understanding-LSTMs/
Gated recurrent unit (GRU)

\[ z_t = \sigma (W_z \cdot [h_{t-1}, x_t]) \]
\[ r_t = \sigma (W_r \cdot [h_{t-1}, x_t]) \]
\[ \tilde{h}_t = \tanh (W \cdot [r_t \ast h_{t-1}, x_t]) \]
\[ h_t = (1 - z_t) \ast h_{t-1} + z_t \ast \tilde{h}_t \]

http://colah.github.io/posts/2015-08-Understanding-LSTMs/
Equivalence of LSTM and GRU

\[ f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \]
\[ i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \]
\[ o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \]
\[ \hat{C}_t = \text{tanh}(W_c[h_{t-1}, x_t] + b_c) \]
\[ C_t = f_t \cdot C_{t-1} + i_t \cdot \hat{C}_t \]
\[ h_t = o_t \cdot \text{tanh}(C_t) \]

\[ z_t = \sigma(W_z \cdot [h_{t-1}, x_t]) \]
\[ r_t = \sigma(W_r \cdot [h_{t-1}, x_t]) \]
\[ \tilde{h}_t = \text{tanh}(W \cdot [r_t \cdot h_{t-1}, x_t]) \]
\[ h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t \]
What you must have learned thus far...

General principles of a recurrent neural network (RNN)

Training an RNN comes with unique challenges:

- Propagating sequences makes it less amenable for parallel implementations
- Vanishing/exploding gradients can be a problem

Variants of a RNN cell using LSTM and GRU

Next class: building a minimal RNN for Language modeling
Thank you!!
ramanathana@ornl.gov