Deep Learning for Text Mining

Part 2. Recurrent Neural Networks (RNN)

Outline

• Optimize Neural Network
  • Stochastic Gradient Descent (Recap)
  • Backpropagation
• Recurrent Neural Network (RNN)
  • Vanilla RNN
  • Gated RNN
Stochastic Gradient Descent (Recap)

- Training data $D = \{(x_i, y_i)\}_{i=1}^n$
- Loss function $l_i(w) = l(f_w(x_i), y_i)$, model $f_w$ parameterized by $w$
- The goal of training: $\min_w \frac{1}{n} \sum_{i=1}^n l_i(w)$
- Random mini-batch: sample $B \sim \text{unif}\{1, 2, \ldots, n\}$
- Update $w$ using the gradient computed from the mini-batch:

$$w^{(k)} := w^{(k-1)} - \eta_k \nabla \left( \frac{1}{|B|} \sum_{i \in B} l_i(w^{(k-1)}) \right)$$

Gradient Computation

- Recall that in logistic regression we use the chain rule as

$$\frac{\partial}{\partial w_j} l_i(w) = \frac{d l_i(w)}{d \sigma(z_i)} \frac{d \sigma(z_i)}{d z_i} \frac{\partial z_i}{\partial w_j}$$

- But, how to compute the gradient for multi-layer neural networks?
Gradient Computation

- Chain rule
  \[ \frac{\partial y}{\partial x} = \frac{\partial y}{\partial h} \frac{\partial h}{\partial x} \]

Gradient Computation (cont’d)

- Multiple-path chain rule
  \[ \frac{\partial y}{\partial x} = \sum_j \frac{\partial y}{\partial h_j} \frac{\partial h_j}{\partial x} \]
Gradient Computation (cont’d)

- Multiple-path chain rule

\[
\frac{\partial y}{\partial x_1} = \sum_j \frac{\partial y}{\partial h_j} \frac{\partial h_j}{\partial x_1}
\]

\[
\frac{\partial y}{\partial x_2} = \sum_j \frac{\partial y}{\partial h_j} \frac{\partial h_j}{\partial x_2}
\]

\[
\frac{\partial y}{\partial x_3} = \sum_j \frac{\partial y}{\partial h_j} \frac{\partial h_j}{\partial x_3}
\]

Gradient Computation (cont’d)

- Vector representation of the gradient

\[
\nabla_x y = \begin{bmatrix} \frac{\partial y}{\partial x_1}, & \frac{\partial y}{\partial x_2}, & \frac{\partial y}{\partial x_3} \end{bmatrix}
\]

\[
= \left( \frac{\partial h}{\partial x} \right)^T \nabla_h y
\]

Jacobian matrix of size $|h| \times |x|$

Gradient vector of size $|h|$
Training Process: An Example

- Model parameters
  \[ W \equiv \{ W_1, W_2, W_3 \} \]
  - \( W_1 \in \mathbb{R}^{h_1 \times |x|} \)
  - \( W_2 \in \mathbb{R}^{h_2 \times h_1} \)
  - \( W_3 \in \mathbb{R}^{h_3 \times h_2} \)

- Training data
  \( D = \{(x_i, y_i) | i = 1, \ldots, n\} \)

- Initialization
  - Assign each parameter a random value

\[ L(y, \hat{y}) = (y - \hat{y})^2 \]
\[ \hat{y} = \text{matmult}(h_2, W_3) \]
\[ h_2 = f(h_1; W_2) \]
\[ h_1 = f(x; W_1) \]

Training Process: An Example (continued)

- Loss function
  \[ L_i(w) \equiv (y_i - \hat{y}_i(w))^2 \]

- Update rule
  \[ W^{(k)} := W^{(k-1)} - \eta_k \frac{1}{|B|} \sum_{i \in B} L_i(W^{(k-1)}) \]

- we need to compute
  \[ \frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2}, \frac{\partial L}{\partial W_3} \]

  e.g. \[ \frac{\partial L}{\partial W_3} = \frac{\partial \hat{y}}{\partial W_3} \frac{\partial L}{\partial \hat{y}} = -2(y - \hat{y}), \frac{\partial \hat{y}}{\partial W_3} = h_2 \]
Training Process: An Example (continued)

- **Forward propagation**
  - Given input $x_i, y_i$ and current $W$, compute the value of each unit (e.g. $L, h_1, h_2, \hat{y}_1$)

- **Backpropagation**
  - Use the values in the forward propagation to compute the gradient, e.g.
    \[
    \frac{dL}{dW_3} = \frac{d\hat{y}}{dW_3} \frac{dL}{d\hat{y}}
    \]
    \[
    \frac{dL}{d\hat{y}} = -2(y_i - \hat{y}_i), \quad \frac{d\hat{y}}{dW_3} = h_2
    \]
  - Update $W$ with the gradient
  - Repeat the above steps iteratively

Outline

- Optimize Neural Network
  - Stochastic Gradient Descent (Recap)
  - Backpropagation
- Recurrent Neural Network (RNN)
  - Vanilla RNN
  - Gated RNN
Vanilla Neural Network

• Hidden layer $h = f(W_{xh}x)$
• Output $\hat{y} = g(W_{yh}h)$
• Example: a binary classifier for text categorization
  • $x$ is a bag-of-words representation of a doc.
  • $g(z) = \tanh(z) \in \{\pm 1\}$, the rescaled logistic sigmoid function
    \[ \sigma(z) \equiv \frac{e^z}{1+e^z} \]
    \[ \tanh(z) \equiv \frac{e^z-e^{-z}}{e^z+e^{-z}} = 2\sigma(2z) - 1 \]

Limitation of Vanilla Neural Network

• Not taking the sequential order of input variables into account
• Not modeling the sequential dependencies among output variables
• Cannot support language modeling, for example, the task of predicting future word(s) based on previously observed ones
Recurrent Neural Network

• Modeling a sequence of \((x_t, h_t, y_t)\) with the recurrence formula for step \(t\) as

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

• Notice the same function \((f)\) and the same set of model parameters \((W)\) are used at every time step.

Specifically, we may define

\[
    \hat{y}_t = g(h_t) \overset{\text{def}}{=} W_{hy} h_t \\
    h_t = f(h_{t-1}, x_t) \overset{\text{def}}{=} \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \\
    \text{for } t = 1, 2, 3, ..., \text{ usually we have } h_0 = \vec{0}
\]
Different RNNs

• Vanilla Neural Network (e.g., for binary image classification)

Different RNNs

• E.g. Image Captioning
  Image → word sequence
Different RNNs

E.g. Text classification
Word sequence $\rightarrow$ Topic label

Different RNNs

E.g. Machine Translation
Word seq. $\rightarrow$ Word seq.
Different RNNs

E.g. POS tagging
Word seq. → Label seq.

Example

Word-level language model

Vocabulary: ["I", "like", "this", "movie"]

Example training sequence: “I like this movie”

It can also be a dense vector trained by word2vec or GloVe
Example

Word-level sentiment classification example

Vocabulary: [“I”, “like”, “this”, “movie”]

Example training sequence: ("I like this movie", Positive)

Output layer

Input layer

Hidden layer

Input words: I like this movie

The vanishing gradient problem

- Multiplying the same matrix at each time step during backpropagation
- Causing the gradient to become very small or very large quickly, which is called the vanishing or exploding gradient
Why do gradients vanish (or explode) in neural nets?

- Example: a multi-layer nnet with a sigmoid function at each layer
- Showing how the gradient of the output variable w.r.t. an input-layer variable would vanish or explode when the number of layers increases

Why do gradients vanish in RNN?

- Suppose the loss function is $L$. We would (incorrectly) compute $\frac{\partial L}{\partial W_{hh}}$ as

$$\frac{\partial L}{\partial W_{hh}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_4} \frac{\partial h_4}{\partial W_{hh}}$$

- But $h_4 = f_W(h_3, x_4)$ depends on $h_3$, which also depends on $W_{hh}$. Thus the correct formula is

$$\frac{\partial L}{\partial W_{hh}} = \sum_{k=1}^{4} \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_4} \frac{\partial h_4}{\partial h_k} \frac{\partial h_k}{\partial W_{hh}}$$

[based on the RNN tutorial by WILDML, 2015]
Why do gradients vanish in RNN (cont’d)

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

Rewrite the gradient as:

\[
\frac{\partial L}{\partial W_{hh}} = \sum_{k=1}^{4} \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_4} \frac{\partial h_4}{\partial h_k} \frac{\partial h_k}{\partial W_{hh}}
\]
\[
= \sum_{k=1}^{4} \frac{\partial L}{\partial \hat{y}} \left( \prod_{j=k+1}^{4} \frac{\partial h_j}{\partial h_{j-1}} \right) \frac{\partial h_k}{\partial W_{hh}}
\]

Multiply the Jacobian matrix multiple times

Why do gradients vanish in RNN (cont’d)

\[
\prod_{j=k+1}^{4} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=k+1}^{4} \frac{\partial \tanh(v_j)}{\partial v_j} \frac{\partial v_j}{\partial h_{j-1}}
\]

Gradient bounded by 1

Notice that \( \frac{\partial h_j}{\partial h_{j-1}} \) is between 0 and 1, could be very near zero -- multiplying it several times would result in a vanishing gradient.
Outline

- Optimize Neural Network
  - Stochastic Gradient Descent (Recap)
  - Backpropagation
- Recurrent Neural Network (RNN)
  - Vanilla RNN
  - Gated RNN

Gated Recurrent Neural Networks

- Addressing the gradient vanishing/exploding issue in naive RNN
- Controlling information flow via gates, which allow some time steps in the sequential process to be skipped
- Representative Approach: Long Short Term Memory (LSTM)
Long Short-Term Memory (LSTM)

- Using “cell state” $c$ (a vector) to control the information flow forward over time
  - It only involves in linear operations and hence is less prone to vanishing gradient (unlike function $tanh$ in the gradient of latent $h$)
- Using “gates” ($f, i, g, o$) to allow context/gradient to pass through without changing

Plain RNN vs. LSTM

- In Vanilla RNN we have
  \[ h_t = \tanh(W_{hh} h_{t-1} + W_{xh} x_t) \]
- We can rewrite it as
  \[ h_t = \tanh(W (h_{t-1} x_t)) \] where \( W = [W_{hh} \ W_{xh}] \)
- LSTM applies the gate functions to vanilla RNN as
  \[
  \begin{bmatrix}
  f \\
  i \\
  o \\
  g
  \end{bmatrix} =
  \begin{bmatrix}
  \sigma \\
  \sigma \\
  \sigma \\
  \tanh
  \end{bmatrix}
  \]
  
  \[ W (h_{t-1} x_t) \]
  
  Same as in plain RNN
LSTM details

• Assume \( x \) and \( h \) have the same dimension \( n \)

\[
\begin{align*}
\text{W} & : 4n \times 2n \\
\text{sigmoid} & : 4n \\
\text{sigmoid} & : 4n \\
\text{tanh} & : 4n \\
\end{align*}
\]

\[
\begin{bmatrix}
W(h_{t-1})x_t \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
i \\
f \\
o \\
g \\
\end{bmatrix}
\]

\[
c_t = f \odot c_{t-1} + i \odot g \\
h_t = o \odot \tanh(c_t)
\]

where \( \odot \) is an element-wise gate operator

• If \( f \) is all ones and \( i \) is all zeros, the current cell ignores the current signals (\( h \) and \( x \) combined in \( g \)) and carries all the memory to the next step.

• What if \( f \) is all zeros and \( i \) is all ones?
LSTM details

Hidden state $h_t$ is the output of tanh activation where outgate $o$ controls $h_t$ going forward not.

- Information flow: Both $c_t$ and $h_t$ carries information to the next time step. But $c_t$ is only updated by linear operations while $h_t$ is updated by a non-linear one.
- GRU (Gated Recurrent Unit) combines $c$ and $h$ into a single latent variable instead two in LSTM (and is more popular).
Performance on language modeling [Yoon Kim et al. AAAI 2016]

<table>
<thead>
<tr>
<th></th>
<th>PPL</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM-Word-Small</td>
<td>97.6</td>
<td>5 m</td>
</tr>
<tr>
<td>LSTM-Char-Small</td>
<td>92.3</td>
<td>5 m</td>
</tr>
<tr>
<td>LSTM-Word-Large</td>
<td>85.4</td>
<td>20 m</td>
</tr>
<tr>
<td>LSTM-Char-Large</td>
<td>78.9</td>
<td>19 m</td>
</tr>
<tr>
<td>KN-5 (Mikolov et al. 2012)</td>
<td>141.2</td>
<td>2 m</td>
</tr>
<tr>
<td>RNN¹ (Mikolov et al. 2012)</td>
<td>124.7</td>
<td>6 m</td>
</tr>
<tr>
<td>RNN-LDA¹ (Mikolov et al. 2012)</td>
<td>113.7</td>
<td>7 m</td>
</tr>
<tr>
<td>genCNN¹ (Wang et al. 2015)</td>
<td>116.4</td>
<td>8 m</td>
</tr>
<tr>
<td>FOFE-FNNLM¹ (Zhang et al. 2015)</td>
<td>108.0</td>
<td>6 m</td>
</tr>
<tr>
<td>Deep RNN (Pascanu et al. 2013)</td>
<td>107.5</td>
<td>6 m</td>
</tr>
<tr>
<td>Sum-Prod Net² (Cheng et al. 2014)</td>
<td>100.0</td>
<td>5 m</td>
</tr>
<tr>
<td>LSTM-1¹ (Zaremba et al. 2014)</td>
<td>82.7</td>
<td>20 m</td>
</tr>
<tr>
<td>LSTM-2¹ (Zaremba et al. 2014)</td>
<td>78.4</td>
<td>52 m</td>
</tr>
</tbody>
</table>

Table 3: Performance of our model versus other neural language models on the English Penn Treebank test set. PPL refers to perplexity (lower is better) and size refers to the approximate number of parameters in the model. KN-5 is a Kneser-Ney 5-gram language model which serves as a non-neural baseline. ¹For these models the authors did not explicitly state the number of parameters, and hence sizes shown here are estimates based on our understanding of their papers or private correspondence with the respective authors.

Reference

- Louis-Philippe Morency, Tadas Baltrusaitis, *CMU 11-777: Advanced Multimodal Machine Learning*
- Fei-Fei Li, Andrej Karpathy, Justin Johnson *Stanford CS231n: Convolutional Neural Networks for Visual Recognition*
- Christopher Olah’s blog: Understanding LSTM Networks
- Denny Britz: *Recurrent Neural Networks tutorial*