NLP Programming Tutorial 8 - Recurrent Neural Nets

Graham Neubig
Nara Institute of Science and Technology (NAIST)
Feed Forward Neural Nets

- All connections point forward

\[ \phi(x) \]

- It is a directed acyclic graph (DAG)
Recurrent Neural Nets (RNN)

• Part of the node outputs return as input

\[ h_{t-1}, \varphi_t(x) \]

• Why? It is possible to “memorize”
RNN in Sequence Modeling

\[ \begin{align*}
  y_1 & \quad \text{NET} \\
  x_1 & \\
  y_2 & \quad \text{NET} \\
  x_2 & \\
  y_3 & \quad \text{NET} \\
  x_3 & \\
  y_4 & \quad \text{NET} \\
  x_4 &
\end{align*} \]
Example: POS Tagging

natural language processing is
Multi-class Prediction with Neural Networks
Review: Prediction Problems

Given \( x \), predict \( y \)

**A book review**
Oh, man I love this book!
This book is so boring...

Is it positive?  
yes  
no

**A tweet**
On the way to the park!
公園に行くなう！

**Its language**
English  
Japanese

**Multi-class Prediction**  
(2 choices)

**A sentence**
I read a book

**Its syntactic parse**

```
  S
 / \  
VP  NP
 /   |
VBD DET NN
```

**Structured Prediction**  
(millions of choices)
Review: Sigmoid Function

- The sigmoid softens the step function

\[ P(y = 1 \mid x) = \frac{e^{w \cdot \varphi(x)}}{1 + e^{w \cdot \varphi(x)}} \]
softmax Function

- Sigmoid function for multiple classes

\[
P(y|x) = \frac{e^{w \cdot \varphi(x, y)}}{\sum_{\tilde{y}} e^{w \cdot \varphi(x, \tilde{y})}}
\]

- Current class

- Sum of other classes

- Can be expressed using matrix/vector ops

\[
r = \exp(W \cdot \varphi(x))
\]

\[
p = \frac{r}{\sum_{\tilde{r} \in r} \tilde{r}}
\]
Selecting the Best Value from a Probability Distribution

- Find the index \( y \) with the highest probability

```python
find_best(p):
    y = 0
    for each element \( i \) in 1 .. len(p)-1:
        if \( p[i] > p[y] \):
            y = i
    return y
```
softmax Function Gradient

• The difference between the true and estimated probability distributions

\[-d \text{err} / d \varphi_{out} = p' - p\]

• The true distribution \(p'\) is expressed with a vector with only the \(y\)-th element 1 (a one-hot vector)

\[p' = \{0, 0, \ldots, 1, \ldots, 0\}\]
Creating a 1-hot Vector

```python
def create_one_hot(id, size):
    vec = np.zeros(size)
    vec[id] = 1
    return vec
```
Forward Propagation in Recurrent Nets
Review: Forward Propagation Code

```python
forward_nn(network, φ₀)

φ = [ φ₀ ]  # Output of each layer

for each layer i in 1 .. len(network):
    w, b = network[i-1]
    # Calculate the value based on previous layer
    φ[i] = np.tanh( np.dot( w, φ[i-1] ) + b )

return φ  # Return the values of all layers
```
RNN Calculation

\[ h_t = \tanh \left( w_{r,h} \cdot h_{t-1} + w_{r,x} \cdot x_t + b_r \right) \]

\[ p_t = \text{softmax} \left( w_{o,h} \cdot h_t + b_o \right) \]
RNN Forward Calculation

```python
forward_rnn(w_r,x, w_r,h, b_r, w_o,h, b_o, x):
    h = [ ] # Hidden layers (at time t)
    p = [ ] # Output probability distributions (at time t)
    y = [ ] # Output values (at time t)
    for each time t in 0 .. len(x)-1:
        if t > 0:
            h[t] = tanh(w_r,x x[t] + w_r,h h[t-1] + b_r)
        else:
            h[t] = tanh(w_r,x x[t] + b_r)
        p[t] = tanh(w_o,h h[t] + b_o)
        y[t] = find_max(p[t])
    return h, p, y
```
Review:
Back Propagation in Feed-forward Nets
Stochastic Gradient Descent

• Online training algorithm for probabilistic models (including logistic regression)

\[
\begin{align*}
    w &= 0 \\
    \text{for } I \text{ iterations} \\
    \quad \text{for each labeled pair } x, y \text{ in the data} \\
    \quad \quad w &+\equiv \alpha \ast \frac{dP(y|x)}{dw}
\end{align*}
\]

• In other words

• For every training example, calculate the gradient (the direction that will increase the probability of \( y \))
• Move in that direction, multiplied by learning rate \( \alpha \)
Gradient of the Sigmoid Function

• Take the derivative of the probability

\[
\frac{d}{dw} P(y = 1 | x) = \frac{d}{dw} \frac{e^{w \cdot \phi(x)}}{1 + e^{w \cdot \phi(x)}} = \varphi(x) \frac{e^{w \cdot \phi(x)}}{(1 + e^{w \cdot \phi(x)})^2}
\]

\[
\frac{d}{dw} P(y = -1 | x) = \frac{d}{dw} \left( 1 - \frac{e^{w \cdot \phi(x)}}{1 + e^{w \cdot \phi(x)}} \right) = -\varphi(x) \frac{e^{w \cdot \phi(x)}}{(1 + e^{w \cdot \phi(x)})^2}
\]
Learning: Don't Know Derivative for Hidden Units!

• For NNs, only know correct tag for last layer

\[ h(x) \]

\[ \frac{d P(y=1|x)}{d w_1} = ? \]

\[ \frac{d P(y=1|x)}{d w_2} = ? \]

\[ \frac{d P(y=1|x)}{d w_3} = ? \]

\[ \frac{d P(y=1|x)}{d w_4} = h(x) \frac{e^{w_4 \cdot h(x)}}{(1 + e^{w_4 \cdot h(x)})^2} \]

\[ y=1 \]
Answer: Back-Propagation

- Calculate derivative w/ chain rule

\[
\frac{d P(y=1|x)}{d w_1} = \frac{d P(y=1|x)}{d w_4 h(x)} \frac{d w_4 h(x)}{d h_1(x)} \frac{d h_1(x)}{d w_1}
\]

\[
e^{w_4 \cdot h(x)} \frac{1}{(1+e^{w_4 \cdot h(x)})^2}
\]

Error of next unit \((\delta_4)\)

Weight Gradient of this unit

In General

Calculate \(i\) based on next units \(j\):

\[
\frac{d P(y=1|x)}{w_i} = \frac{d h_i(x)}{d w_i} \sum_j \delta_j w_{i,j}
\]
Conceptual Picture

- Send errors back through the net
Back Propagation in Recurrent Nets
What Errors do we Know?

- We know the output errors $\delta_o$
- Must use back-prop to find recurrent errors $\delta_r$
How to Back-Propagate?

- Standard *back propagation through time (BPTT)*
  - For each $\delta_o$, calculate $n$ steps of $\delta_r$
- Full gradient calculation
  - Use dynamic programming to calculate the whole sequence
Back Propagation through Time

- Use only one output error
- Stop after n steps (here, n=2)
Full Gradient Calculation

- First, calculate whole net result forward
- Then, calculate result backwards
BPTT? Full Gradient?

- Full gradient:
  - + Faster, no time limit
  - - Must save the result of the whole sequence in memory

- BPTT:
  - + Only remember the results in the past few steps
  - - Slower, less accurate for long dependencies
Vanishing Gradient in Neural Nets

- "Long Short Term Memory" is designed to solve this
RNN Full Gradient Calculation

```python
def gradient_rnn(w_rx, w_rh, br, wo,h, bo, x, h, p, y_prime):
    # Initialize gradients
    Δwr,x, Δwr,h, Δbr, Δwo,h, Δbo = initialize Δwr,x, Δwr,h, Δbr, Δwo,h, Δbo

    δ_r_prime = np.zeros(len(br))  # Error from the following time step
    for each time t in len(x)-1 .. 0:
        p_prime = create_one_hot(y_prime[t])
        δ_o_prime = p_prime - p[t]  # Output error
        Δwo,h += np.outer(h[t], δ_o_prime); Δbo += δ_o_prime  # Output gradient
        δ_r = np.dot(δ_r_prime, w_rh) + np.dot(δ_o_prime, wo,h)  # Backprop
        δ_r_prime = δ_r * (1 - h[t]**2)  # tanh gradient
        Δwr,x += np.outer(x[t], δ_r_prime); Δbr += δ_r_prime  # Hidden gradient
        if t != 0:
            Δwr,h += np.outer(h[t-1], δ_r_prime);
    return Δwr,x, Δwr,h, Δbr, Δwo,h, Δbo
```

30
Weight Update

```python
update_weights(w_{r,x}, w_{r,h}, b_r, w_{o,h}, b_o, \Delta w_{r,x}, \Delta w_{r,h}, \Delta b_r, \Delta w_{o,h}, \Delta b_o, \lambda)

w_{r,x} += \lambda \times \Delta w_{r,x}
w_{r,h} += \lambda \times \Delta w_{r,h}
b_r += \lambda \times \Delta b_r
w_{o,h} += \lambda \times \Delta w_{o,h}
b_o += \lambda \times \Delta b_o
```
# Create features
create map \( x_{ids}, y_{ids} \), array \( data \)
for each labeled pair \( x, y \) in the data
    add \( (create\_ids(x, x_{ids}), create\_ids(y, y_{ids})) \) to \( data \)
initialize \( net \) randomly

# Perform training
for \( I \) iterations
    for each labeled pair \( x, y' \) in the \( feat\_lab \)
        \( h, p, y = forward\_rnn(net, \varphi_0) \)
        \( \Delta = gradient\_rnn(net, x, h, y') \)
        update_weights\( (net, \Delta, \lambda) \)

print \( net \) to \( weight\_file \)
print \( x_{ids}, y_{ids} \) to \( id\_file \)
Exercise
Exercise

- Create an RNN for sequence labeling
- Training train-rnn and testing test-rnn
- **Test**: Same data as POS tagging
  - Input: test/05-{train,test}-input.txt
  - Reference: test/05-{train,test}-answer.txt
- **Train** a model with data/wiki-en-train.norm_pos and **predict** for data/wiki-en-test.norm
- **Evaluate** the POS performance, and **compare** with HMM:
  script/gradeapos.pl data/wiki-en-test.pos my_answer.pos