Notes on RNN
Siemng Sun (simsun@seas.upenn.edu)
Last update: May, 2018

1 BPTT and Gradient vanishing/exploding

I learned a lot from https://github.com/oxford-cs-deepnlp-2017/lectures This note is based a lot on lecture 7.

Language modeling is one of the most common task RNN can be applied. The objective function of language model is to maximize the log probability of a sentence, or minimize negative log likelihood. Assume the loss on one sentence of length $N$ is

$$l(\theta) = -\sum_{i=1}^{N} \log p(x_i; \theta)$$

At each time step, a distribution over vocabulary is predicted, $p(x_i)$ stands for the probability the correct word $x_i$ is assigned to. Suppose we have an one-layer RNN, for each timestep, the output is a distribution of words in the vocabulary. Let $\hat{y}_t$ be the predicted output, $y_t$ be the one hot vector of the word on step $t$, $x_t$ be the input on that step, $h_t$ be the hidden state computed after step $t$, then

$$\hat{y}_t = \sigma(Wf(V[h_{t-1}; x_t] + b) + b_0)$$  \hspace{1cm} (1)$$

$$h_t = f(V[h_{t-1}; x_t] + b)$$  \hspace{1cm} (2)

At step $t$, the hidden state of an ideally trained model is supposed to encode all past information rendered before $t^{th}$ step. Therefore, for a sentence of $N$ words, $h_N$ is a continuous representation of this sentence, which is similar to the feature vector/matrix we get after convolution and pooling layers in a CNN. Back propagation when applied on time series problem is called back prop through time (BPTT); for very long sequences, BPTT is sometimes extended to truncated BPTT(TBPTT) to mitigate gradient vanishing. Gradient clipping can be applied to avoid gradient exploding.

A question we might want to ask is: how does the loss on a particular step is going to affect all former decisions. To simplify the answer, we will look on the loss on the last step and the hidden state at step $t$. Let $z$ be the linear operation inside function $f$, $l_T$ be the loss on the last time step,

$$z = V_x x_t + V_h h_{t-1}$$

$$\frac{\partial l_T}{\partial h_t} = \frac{\partial l_T}{\partial \sigma} \frac{\partial \sigma}{\partial f} \frac{\partial f}{\partial h_T} \prod_{i \in \{T-1, \ldots, t+1\}} \frac{\partial h_i}{\partial h_{i-1}}$$  \hspace{1cm} (2)$$

$$= \frac{\partial l_T}{\partial \sigma} \frac{\partial \sigma}{\partial f} \frac{\partial f}{\partial h_T} \prod_{i \in \{T-1, \ldots, t+1\}} \frac{\partial h_i}{\partial z_i} \frac{\partial z_i}{\partial h_{i-1}}$$  \hspace{1cm} (3)$$

$$= \frac{\partial l_T}{\partial \sigma} \frac{\partial \sigma}{\partial f} \frac{\partial f}{\partial h_T} \prod_{i \in \{T-1, \ldots, t+1\}} \frac{\partial h_i}{\partial z_i} V_h$$  \hspace{1cm} (4)$$

$$= C(V_h)^{T-t-1}$$  \hspace{1cm} (5)$$
f is a shorthand for \( f(V[h_{t-1};x_t] + b) \), so are \( \sigma \) and \( l_T \) shorthands. In (3), I merge all other factors to \( C \). Assume \( \lambda \) is an eigenvalue of \( V_h \) and \( x \) the corresponding eigenvector, then

\[
V_h x = \lambda x
\]

\[
(V_h)^{T-t-1} x = \lambda^{T-t-1} x
\]

- if \( \lambda = 1 \), then \( \frac{\partial l_T}{\partial h_t} \) is fine;
- if \( \lambda > 1 \), gradient explodes
- if \( \lambda < 1 \), gradient vanishes.

**TBPTT** During training, the loss on step \( t \) does not propagate back to step 0, instead it only flows back to the maximum chunk size we set, so the gradient computation becomes faster, the value of gradient neither becomes too large nor too small. However, if a pattern (correlated words in a phrase) is divided to 2 chunks all the time through training data, then we might not be able to correctly learn the correlation.

## 2 Conditional Language Model

The sentence probability of an unconditional n-gram language model can be written as

\[
p(w) = \prod_{t=n}^N p(w_t | w_{t-n-1})
\]

If we need to generate text based on a sequence of input, such as machine translation, text summarization and question answering, the probability of output sequence \( w \) given some input \( x \), without n-gram history constrain, can be written as

\[
p(w | c) = \prod_{t=1}^T p(w_t | c, w_{1:t-1})
\]

For an encoder-decoder structure, \( c \) is a compressed and usually continuous representation of input sequence. It can be the last hidden state of input RNN, or the last cell state and hidden state of input LSTM cell. In each decoding step, except for receiving the predicted word from last step as input, it also takes in the compressed vector \( c \). Thus the decoding RNN can be extended from eq.1

\[
\hat{y}_t = \sigma(Wf(V[h_{t-1};x_t] + V_c + b) + b_0)
\]

During training phase, on step \( t \), the decoder takes in the last hidden state \( h_{t-1} \) and the last predicted output \( \hat{y}_{t-1} \) to predict current \( \hat{y}_t \). An erroneous prediction at \( t^{th} \) step may strongly influence the later predictions, therefore the loss on step \( t + k \), \( -\log p(\hat{y}_t; \theta) \), can be very high. Without enough guidance on decoding steps during training, large loss leads to unstable states and thus makes it hard to converge.

## 3 Curriculum learning

**Teacher forcing** is a training approach to speed up convergence for RNNs. What teacher forcing does is simply feed the correct label \( y_{t-1} \) to the next step instead of using the possibly wrongly predicted \( \hat{y}_{t-1} \).

**Curriculum learning** In Bengio’s paper, they described curriculum as a series of training criterion associated with different set of weights on training examples. At first, simple examples are assigned to higher weights, hard examples are gradually assigned to higher weights. let \( z \) be a pair of example, \( Q_\lambda(z) \) is the distribution of \( z \) after assigned to weights at curriculum \( \lambda \), such that \( \int Q_\lambda(z) = 1 \), where
$0 \leq \lambda \leq 1$. Overall, this idea is very similar to adaboost where after every iteration, the incorrectly predicted examples (hard examples) are assigned with higher weights.

**Curriculum learning and encoder-decoder** Sometimes it’s hard to decide whether an example is hard to predict or not. In real training phase of sequence modeling, we can start with simple examples which we use in **teacher forcing** – the loss can only reflect the loss on individual decoding steps (the losses on each steps are not related). Then we can randomly select examples as hard examples by making it use the $\hat{y}_{t-1}$ as input and gradually increase the number of hard examples.