How LSTM solves the vanishing gradient problem of RNN

1. Introduction

RNN enables the modeling of time-dependent and sequential data tasks effectively, such as clinical prediction, machine translating , text generation, speech recognition, image description generation and etc.

In traditional neural networks, we assume that neurons in the same layers are independent, which has led to poor performance in some tasks. For example, if you want to predict the next word in a sentence in a language model, it is better to know which the previous words are, cause next word is dependent on the previous ones.

The main difference between RNN and traditional neural networks is its feedback loop, i,e. the output of hidden layer in previous timestep will feedback to the same layer. If you are not familiar with RNN, you can click this tutorial : Recurrent Neural Networks Tutorial

In theory, RNN can use information in arbitrarily long sequences, but in practice they are limited to a few steps back due to the vanishing gradient problem, which hinder it from learning longer data sequences.

So what is the vanishing gradient problem? Why gradient vanishing is a problem? How to compute neural network gradient in a completely vectorized way? What is the vanishing problem of RNN and how to overcome the vanishing gradient problem by using Long Short Term Memory networks (LSTM) ?

With these questions, we look at each one.

2. Vanishing gradient problem

2.1 Gradient descent

In neural network, the input data is transferred forward through the network to get the output. Based on the difference between the output and the expected output, we calculated the loss function $J(\theta)$, $\theta \in \mathbb{R}^l$. For iterative minimization of the loss function $J(\theta)$, a widely used method is $gradient\ descent$. To begin with, we initialize the parameter $\theta^{(\mathbf{0})}$, and then at the ith iteration:

$$\theta^i = \theta^{(i-1)} + \mu_i \Delta\theta^{(i)}, i > 0$$

In this formula, μ_i is known as the $step\ length$, and $\mu_i > 0$. $\Delta\theta^{(i)}$ is known as $update\ direction$. In the gradient descent method, the choice of $\Delta\theta^{(i)}$ is done to guarantee that:

$$J(\theta^{(i)}) < J(\theta^{(i-1)})$$

Assume that at the $i-1$ step iteration, the value $\theta^{(i-1)}$ has obtained, then applying the first order Taylor's expansion around $\theta^{(i-1)}$, we have:

$$J(\theta^{(i)}) = J(\theta^{(i-1)} + \mu_i \Delta\theta^{(i)}) \approx J(\theta^{(i-1)}) + \mu_i \bigtriangledown^T J(\theta^{(i-1)} \Delta\theta^{(i)})$$

To guarantee $J(\theta^{(i)}) < J(\theta^{(i-1)})$, we must let:

$$\bigtriangledown^T J(\theta^{(i-1)}\Delta\theta^{(i)} < 0$$

For such a choice, $\Delta\theta^{(i)}$ and $\bigtriangledown J(\theta)^{(i-1)}$ must form an obtuse angle. The geometry is illustrated in below figures, the left figure represents a cost function in the two dimensional parameter space, while the right figure stands for $\Delta\theta^{(i)}$ and $\bigtriangledown J(\theta)^{(i-1)}$ form an obtuse angle.

Note: The gradient vector $\bigtriangledown J(\theta)^{(i-1)}$ is perpendicular to the tangent plane at the isovalue curve at the point $\theta^{(i-1)}$.

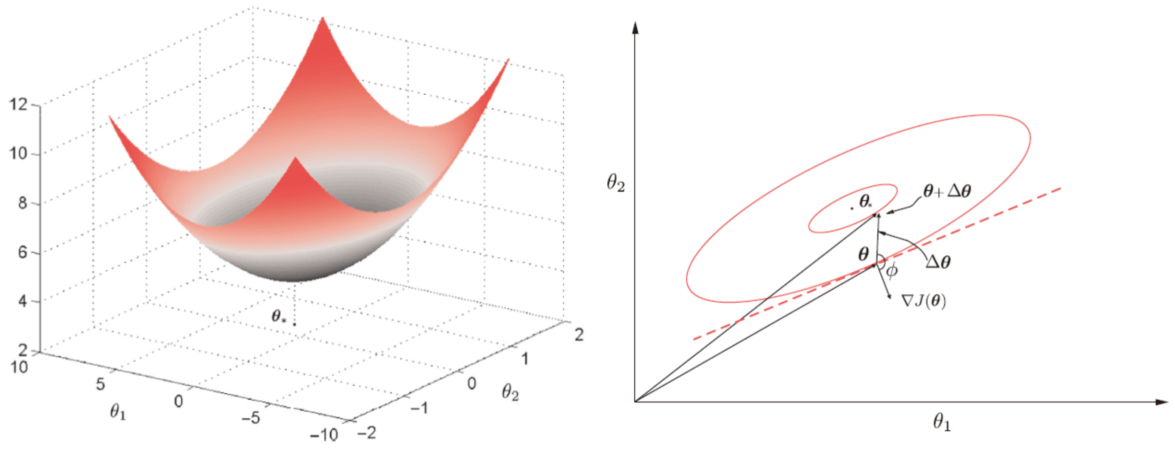


Image Source: [3], from page 164 to page 165.

As we analyzed before, when it comes to select the search direction $\Delta\theta^{(i)}$ to guarantee $J(\theta^{(i)}) < J(\theta^{(i-1)})$, we should make $\Delta\theta^{(i)}$ and $\bigtriangledown J(\theta)^{(i-1)}$ form an obtuse angle, but which is the best search direction to move?

Let us assume $\mu_i = 1$ and search for all vectors, $\mathbf{z}$, with unit Euclidean norm, $\theta^{(i-1)}$. It is obvious that the best search direction is the one that gives the most negative value of the inner product, i. e. $\bigtriangledown^T J(\theta^{(i-1)}\mathbf{z} = -1$. So:

$$z = -\frac{\bigtriangledown J(\theta^{(i-1)})}{\|\bigtriangledown J(\theta^{(i-1)})\|}$$

Thus, the best and steepest descent direction is consistent with negative gradient direction, and the parameter update formula can be rewrited as:

$$\theta^{(i)} = \theta^{(i-1)} - \mu_i \bigtriangledown J(\theta^{(i-1)})$$

2.2 What is gradient vanishing problem and why gradient vanishing is a problem?

As we have derived:

$$\theta^{(i)} = \theta^{(i-1)} - \mu_i \bigtriangledown J(\theta^{(i-1)})$$

In each iteration of training, the neural network's weights receives an update proportional (μ_i) to the gradient $J(\theta^{(i-1)}$. The problem is that in some cases the gradient becomes vanishingly small, preventing the weights being updated.

Actually, in deep neural networks, the gradients calculated by each neuron will be passed to previous layer due to the chain rule. With the number of layers increases, the gradient update information calculated will decay exponentially and the gradient will disappear. Take a simple example, if we multiply 0.9 twice, we will get 0.81. What if we multiply 100 times? Then we will get a number close to 0.

Similarly, In RNN, Vanishing gradient problem hinder it from learning longer data sequences. To understand the effect of vanishing gradient problem, let's take an intuitive example in language model. If we want to predict the next word of a sentence: $\mathit{The\ writer\ of\ the\ books\_}$. Actually, the next word is dependent on the writer, so the correct word is is, but the output is most likely be are.

> Due to vanishing gradient problem, RNN-LMs are better at learning from sequential recency that syntactic recency, so they make this type of error more often than we'd like[4,6]

On the contrary, if the gradient becomes too big, then gradient descent update step becomes too big, leading to the network unstable. In the worst case, the weight can even overflow and become a value that can never be updated.

3. How to compute Neural Network Gradients

3.1 Vectorized Gradients

Suppose we have a function $\mathbf{f}:\mathbb{R}^{\mathrm{n}} \to \mathbb{R}^{\mathrm{m}}$ which maps a vector of length n to a vector of length m: $\mathbf{f}(\mathbf{x}) = [f_1(x_1, x_2, \ldots x_n), f_2(x_1, x_2, \ldots x_n), \ldots f_m(x_1, x_2, \ldots x_n)]$. Then its Jacobian is the following $m \times n$ matrix:

$$\frac{\partial \mathbf{f}}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \vdots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

That is ,$(\frac{\partial f}{\partial x})_{ij} = \frac{\partial f_i}{\partial x_j}$ (which is just a standard non-vector derivative). The Jacobian matrix will be useful for us because we can apply the chain rule to a vector-valued function just by multiplying Jacobian.

3.2 Useful Identities

This part will introduce how to compute Jacobian of several simple functions which will be used in taking neural network gradients. For details, click: Computing Neural Network Gradients.

(1) Matrix times column vector with respect to the column vector ($\mathbf{z} = \mathbf{W}\mathbf{x}$, what is $\frac{\partial \mathbf{z}}{\partial \mathbf{x}}$?)

Suppose that $\mathbf{W} \in \mathbb{R}^{n \times m}$. We can think $\mathbf{z}$ as a function of $\mathbf{x}$ taking an m-dimensional vector to an n-dimensional vector. So its Jacobian will be $n \times m$. According to the rule of matrix multiplication, we can easily know :

$$z_i = \sum_{k=1}^{m} W_{ik} x_k$$

So an entry $(\frac{\partial f}{\partial x})_{ij}$ of the Jacobian will be

$$(\frac{\partial f}{\partial x})_{ij} = \frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j} \sum_{k=1}^{m} W_{ik} x_k = \sum_{k=1}^{m} \frac{\partial}{\partial x_j} x_k = W_{ij}$$

So we can see that $\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \mathbf{W}$

(2) A vector with itself($\mathbf{z} = \mathbf{x}$, what is $\frac{\partial \mathbf{z}}{\partial \mathbf{x}}$?)

Since $z_i = x_i$. So

$$(\frac{\partial f}{\partial x})_{ij} = \frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j} x_i = \begin{cases} 1 & \text{if } i = j. \\ 0 & \text{if otherwise.} \end{cases}$$

So we can see that the Jacobian $\frac{\partial \mathbf{z}}{\partial \mathbf{x}}$ is a diagonal matrix where the entry at (i, i) is 1. This is just the identity matrix. $\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \mathbf{I}$

(3) An elementwise function applied to a vector ($\mathbf{z} = f(\mathbf{x})$, what is $\frac{\partial \mathbf{z}}{\partial \mathbf{x}}$?)

Since $z_i = f(x_i)$. So

$$(\frac{\partial f}{\partial x})_{ij} = \frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j} f(x_i) = \begin{cases} f'(x) & \text{if } i = j. \\ 0 & \text{if otherwise.} \end{cases}$$

So we can see that the Jacobian $\frac{\partial \mathbf{z}}{\partial \mathbf{x}}$ is a diagonal matrix where the entry at (i, i) is the derivative of f applied to x_i. So we can write this as $\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = diag(f'(x))$

(4)A scalar J with respect to the matrix $\mathbf{W}$? (Introduce an intermediate variable $\mathbf{z}$, and $\mathbf{z} = \mathbf{W}\mathbf{x}, \delta = \frac{\partial J}{\partial \mathbf{z}}$, what is $\frac{\partial J}{\partial \mathbf{W}} = \frac{\partial J}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{W}} = \delta \frac{\partial \mathbf{z}}{\partial \mathbf{W}}$?)

While taking gradients, we must clear the dimensions of each parameters. J is a scalar, while $\mathbf{z}$ is a n-dimensional vector, so the Jacobian $\delta = \frac{\partial J}{\partial z}$ is a $1 \times n$ vector. Suppose that $\mathbf{W} \in \mathbb{R}^{n \times m}$, $\frac{\partial \mathbf{z}}{\partial \mathbf{W}}$ will be an $n \times n \times m$ tensor which is quite complicated. However, we can take the gradient with respect to a single weight W_{ij}. $\frac{\partial \mathbf{z}}{\partial W_{ij}}$ is just a $n \times 1$ vector, which is much easier to deal with. We have

$$z_k = \sum_{l=1}^{m} W_{kl} x_l$$

$$\frac{\partial z_k}{\partial W_{ij}} = \sum_{l=1}^{m} x_l \frac{\partial}{\partial W_{ij}} W_{kl} = \begin{cases} x_j & \text{if } i = k \text{ and } j = l. \\ 0 & \text{if otherwise.} \end{cases}$$

Another way of writing this is

$$\frac{\partial \mathbf{z}}{\partial W_{ij}} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ x_j \\ 0 \\ \vdots \\ 0 \end{bmatrix} \leftarrow i\text{th element}$$

Now let's compute $\frac{\partial J}{\partial W_{ij}}$

$$\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial W_{ij}} = [\, \delta_1 \quad \cdots \quad \delta_i \quad \cdots \quad \delta_n \,] \begin{bmatrix} 0 \\ \vdots \\ 0 \\ x_j \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \delta_i x_j$$

To get $\frac{\partial J}{\partial \mathbf{W}}$ we want a matrix where entry (i, j) is $\delta_i x_j$.

This matrix is equal to the outer product. So $\frac{\partial J}{\partial \mathbf{W}} = \delta^T x^T$

These basic identities will be enough for us to compute the gradients for RNN and LSRM networks.

4. The vanishing/exploding gradient problem of RNN:

As we mentioned before, The main difference between RNN and traditional neural networks is its feedback loop, i,e. the output of hidden layer in previous timestep will feedback to the same layer.

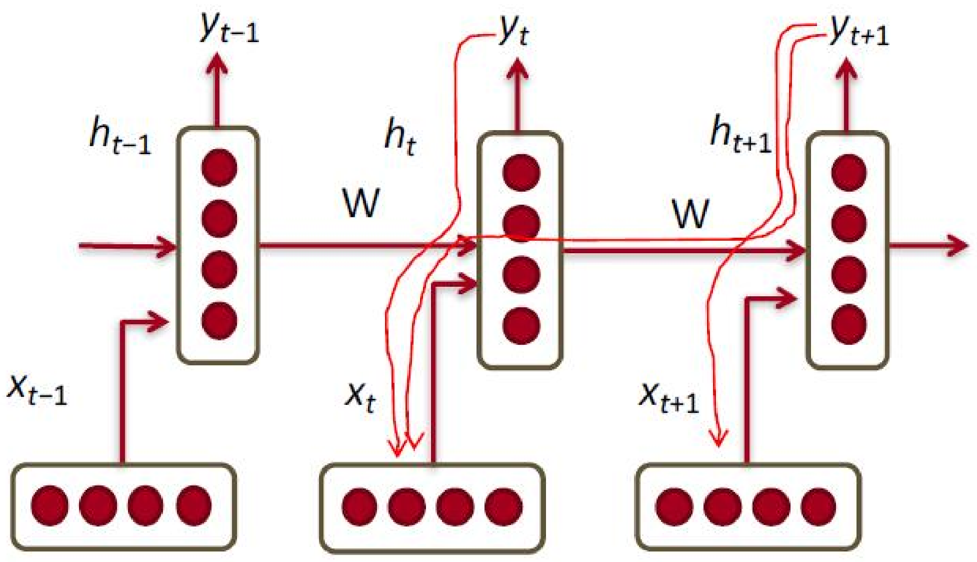


Image Source: CS224n, Stanford

The hidden vector and the output is computed as such:

$$\mathbf{h}_t = \sigma(\mathbf{W}\mathbf{h}_{t-1} + \mathbf{W}^{\mathbf{hx}}\mathbf{x}_t)$$
$$\hat{\mathbf{y}}_{\mathbf{t}} = \mathbf{W}^{(\mathbf{S})}\mathbf{h}_t$$

Notations:

- $\mathbf{x}_t \in \mathbb{R}^d$: the input vectors at time t
- $\mathbf{W}^{\mathbf{hx}} \in \mathbb{R}^{D_h \times d}$: weights matrix used to condition the input vector, x_t
- $\mathbf{W} \in \mathbb{R}^{D_h \times D_h}$: weights matrix used to condition the output of the previous time-step, h_{t-1}
- $\mathbf{h}_{t-1} \in \mathbb{R}^{D_h}$: output of the non_linear function at the previous time-step , $t-1$. $h_0 \in \mathbb{R}^{D_h}$ is an initialization vector for the hidden layer at time-step $t = 0$.
- $\sigma()$: the non-linearity activation function (sigmoid here)
- $\mathbf{y}_t \in \mathbb{R}^V$: the output vectors at time t
- $\mathbf{W}^{\mathbf{S}} \in \mathbb{R}^{V \times D_h}$

To do backpropagation through time to update weight parameters , we need to compute the gradient of error with respect to $\mathbf{W}$. Total error is the sum of each error at time step t, as $\mathbf{W}$ is fixed all the time steps, so the overall error gradient is:

$$\frac{\partial E}{\partial \mathbf{W}} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial \mathbf{W}}$$

So what is the derivative of E_t with respect to the repeated weight matrix ? Just apply the multivariable chain rule:

$$\frac{\partial E_t}{\partial \mathbf{W}} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial \mathbf{W}}\Big|_k \frac{\partial \mathbf{W}\big|_k}{\partial \mathbf{W}} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial \mathbf{W}}\Big|_k$$

It can be further written as:

$$\frac{\partial E_t}{\partial \mathbf{W}} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial \mathbf{W}}\Big|_k = \sum_{k=1}^{t} \frac{\partial E_t}{\partial \mathbf{y_t}} \frac{\partial \mathbf{y_t}}{\partial \mathbf{h_t}} \frac{\partial \mathbf{h_t}}{\partial \mathbf{h_k}} \frac{\partial \mathbf{h_k}}{\partial \mathbf{W}}$$

For the term $\frac{\partial \mathbf{h_t}}{\partial \mathbf{h_k}}$, we use another chain rule application to compute it

$$\frac{\partial \mathbf{h_t}}{\partial \mathbf{h_k}} = \frac{\partial \mathbf{h_t}}{\partial \mathbf{h_{t-1}}} \frac{\partial \mathbf{h_{t-1}}}{\partial \mathbf{h_{t-2}}} \cdots \frac{\partial \mathbf{h_{k+1}}}{\partial \mathbf{h_k}} = \prod_{j=k+1}^{t} \frac{\partial \mathbf{h_j}}{\partial \mathbf{h_{j-1}}}$$

Thus, we have

$$\frac{\partial E}{\partial \mathbf{W}} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial E_t}{\partial \mathbf{y_t}} \frac{\partial \mathbf{y_t}}{\partial \mathbf{h_t}} \Big(\prod_{j=k+1}^{t} \frac{\partial \mathbf{h_j}}{\partial \mathbf{h_{j-1}}} \Big) \frac{\partial \mathbf{h_k}}{\partial \mathbf{W}}$$

As the hidden vector is computed as

$$\mathbf{h_t} = \sigma(\mathbf{W}\mathbf{h_{t-1}} + \mathbf{W^{hx}}\mathbf{x_{[t]}})$$

Using the chain and identities in part 2 we have derived, therefore :

$$\frac{\partial \mathbf{h_j}}{\partial \mathbf{h_{j-1}}} = diag(\sigma'(\mathbf{W}\mathbf{h_{t-1}} + \mathbf{W^{hx}}\mathbf{x_{[t]}}))\mathbf{W}$$

As the paper On the difficulty of training Recurrent Neural Networks has shown:

> To understand the vanishing gradient phenomenon, we need to look at the form of each temporal component, and in particular the matrix factors $\frac{\partial h_t}{\partial h_k}$ that take the form of a product of $t-k$ Jacobian matrices. (In the same way a product of $t-k$ real numbers can shrink to zero or explode to infinity, so does this product of matrices).

Consider matrix L_2 norms, we have

$$\left\| \frac{\partial \mathbf{h_j}}{\partial \mathbf{h_{j-1}}} \right\| \leq \left\| diag(\sigma'(\mathbf{W}\mathbf{h_{t-1}} + \mathbf{W^{hx}}\mathbf{x_{[t]}})) \right\| \|\mathbf{W}\|$$

So,

$$\left\| \frac{\partial \mathbf{h_t}}{\partial \mathbf{h_k}} \right\| = \prod_{j=k+1}^{t} \frac{\partial \mathbf{h_j}}{\partial \mathbf{h_{j-1}}} \leq \|\mathbf{W}\|^{(t-k)} \prod_{j=k+1}^{t} \left\| diag(\sigma'(\mathbf{W}\mathbf{h_{t-1}} + \mathbf{W^{hx}}\mathbf{x_{[t]}})) \right\|$$

The value of $\sigma'(\mathbf{W}\mathbf{h_{t-1}} + \mathbf{W^{hx}}\mathbf{x_{[t]}})$is the derivation of activation function, so it can only be as large as 1 given the sigmoid non-linearity function. In the paper On the difficulty of training Recurrent Neural Networks, Pascanu et al. showed that if the largest eigenvalue of $\mathbf{W}$ is less than 1, then the gradient will shrink exponentially.

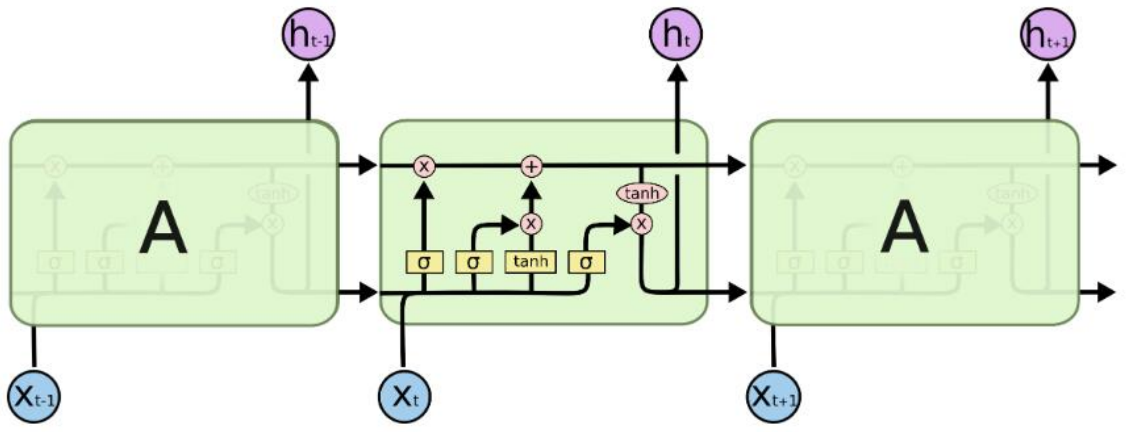
The consequence of gradient vanishing is that the gradient from faraway is lost, and the model weights are only updated with respect to near effects, not long-term effects. There is also another perspective that gradient can be viewed as a measure of the effect of the past on the future.

There is a similar proof relating a largest eigenvalue larger than 1, the gradient explodes. If the gradient becomes too big, then the SGD update step becomes too big , causing the divergence of network training. However, we can clip the gradient if the norm of the gradient is greater than some threshold.

5. How LSTM solves the vanishing gradient problem?

As we can see above, the consequence of gradient vanishing is that the weights are updated only with respect to near effects, not long-term effects. So is there any possible ways to solve this problem? One possible way is to replace the sigmoid or tanh activation function to relu activation function or initialize the weight matrix correctly. Another possible way is using the LSTM netwoks.

Long Short-Term Memory, a type of RNN, is proposed by Hochreiter and Schmidhuber in 1997 as a solution to the vanishing gradients problem. The paper is here: LSTM



The repeating module in an LSTM contains four interacting layers.

Image Source:http://colah.github.io/posts/2015-08-Understanding-LSTMs/

We have a sequence of input $\mathbf{x}_t$, and we will compute a sequence of hidden states $\mathbf{h}_t$ and cell states $\mathbf{c}_t$, both are vectors of length n.

$$\begin{aligned}
\mathbf{f}_t &= \sigma(\mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{U}_f \mathbf{x}_t) \\
\mathbf{i}_t &= \sigma(\mathbf{W}_i \mathbf{h}_{t-1} + \mathbf{U}_i \mathbf{x}_t) \\
\mathbf{o}_t &= \sigma(\mathbf{W}_o \mathbf{h}_{t-1} + \mathbf{U}_f \mathbf{x}_t)
\end{aligned}$$

$$\begin{aligned}
\tilde{\mathbf{c}}_t &= tanh(\mathbf{W}_c \mathbf{h}_{t-1} + \mathbf{U}_c \mathbf{x}_t) \\
\mathbf{c}_t &= \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \tilde{\mathbf{c}}_t \\
\mathbf{h}_t &= \mathbf{o}_t \circ tanh(\mathbf{c}_t)
\end{aligned}$$

Note: $\circ$ represents element-wise product. That means gates are applied using element-wise product.

On time t :

- $\mathbf{f}_t$: **Forget gate**, controls what part of previous cell state are forgotten or kept.
- $\mathbf{i}_t$: **Input gate**, controls what part of the new cell content are written to the cell.
- $\mathbf{o}_t$: **Output gate**, controls what part of cell are output to hidden state.
- $\tilde{\mathbf{c}}_t$: **New cell content** to be written to the cell
- $\mathbf{c}_t$: **Cell state.** Erase ("forget") some content from the last cell state, and write ("input") some new cell content.
- $\mathbf{h}_t$: Read ("output") some content from the cell.

As mentioned before, the reason why RNN has vanishing gradient problem is the recursive derivative $\frac{\partial \mathbf{h_j}}{\partial \mathbf{h_{j-1}}}$, which is fixed for all time step t . if the value is greater than 1, the gradient exploding occurs. On the contrary, if the value is less than 1, the gradient vanishing occurs.

In LSTM, we also calculated the recursive derivative $\frac{\partial \mathbf{c_j}}{\partial \mathbf{c_{j-1}}}$ to loot at what is the main differences compared to RNN.

As the LSTM equation shows: $\mathbf{c_j}$ is the function of $\mathbf{f}_j$ (forget gate), $\mathbf{i}_j$ (input gate), $\tilde{\mathbf{c}}_j$, and each is the function of $\mathbf{h}_{j-1}$ and further, $\mathbf{h}_{j-1}$is the function of $\mathbf{c}_{j-1}$. Applying the multivariable chain rule, we will have:

$$\frac{\partial \mathbf{c}_j}{\partial \mathbf{c}_{j-1}} = \frac{\partial \mathbf{c}_j}{\partial \mathbf{f}_j}\frac{\partial \mathbf{f}_j}{\partial \mathbf{h}_{j-1}}\frac{\partial \mathbf{h}_{j-1}}{\partial \mathbf{c}_{j-1}} + \frac{\partial \mathbf{c}_j}{\partial \mathbf{c}_{j-1}} + \frac{\partial \mathbf{c}_j}{\partial \mathbf{i}_j}\frac{\partial \mathbf{i}_j}{\partial \mathbf{h}_{j-1}}\frac{\partial \mathbf{h}_{j-1}}{\partial \mathbf{c}_{j-1}} \\ + \frac{\partial \mathbf{c}_j}{\partial \tilde{\mathbf{c}}_\mathbf{j}}\frac{\partial \tilde{\mathbf{c}}_\mathbf{j}}{\partial \mathbf{h}_{j-1}}\frac{\partial \mathbf{h}_{j-1}}{\partial \mathbf{c}_{j-1}}$$

$$\begin{aligned}\frac{\partial \mathbf{c}_j}{\partial \mathbf{c}_{j-1}} &= diag(\mathbf{c}_{t-1})diag(\sigma'(\cdot))\mathbf{W}_h * \mathbf{o}_{j-1}diag(tanh'(\mathbf{c}_{j-1})) \\ &+ f_j \\ &+ diag(\tilde{\mathbf{c}}_j)diag(\sigma(\cdot))\mathbf{W}_i * \mathbf{o}_{j-1}diag(tanh'(\mathbf{c}_{j-1})) \\ &+ diag(\mathbf{i}_j)diag(tanh(\cdot))\mathbf{W}_o * \mathbf{o}_{j-1}diag(tanh'(\mathbf{c}_{j-1}))\end{aligned}$$

So as to RNN, if we want to backpropagate back k times, we just multiply this term k times to update the parameters.

In RNN, at any step j, the recursive derivative $\frac{\partial \mathbf{h_j}}{\partial \mathbf{h_{j-1}}}$ is fixed and can only be one value. However, in LSTM, the recursive derivative $\frac{\partial \mathbf{c_j}}{\partial \mathbf{c_{j-1}}}$ can take different values with the different value of f_j, so it can solves the gradient vanishing problem to a certain degree.

LSTM doesn't guarantee that there is no vanishing/exploring gradient, but it does provide an easier way for the model to learn long-distance dependencies.

Last but not least, vanishing/exploding gradient problem is not just a problem for RNN. Actually, due to the chain rule and the choice of nonlinearity activation function, gradient can become vanishingly small or explodingly large as it backpropagates, expecially in deep neural networks.

Reference

[1] Recurrent Neural Networks Tutorial

[2] Understanding LSTM Networks

[3] Theodoridis S. Machine learning: a Bayesian and optimization perspective[M]. Academic Press, 2015.

[4] CS224n: Natural Language Processing with Deep Learning

[5] Why LSTMs Stop Your Gradients From Vanishing: A View from the Backwards Pass

[6] Linzen T, Dupoux E, Goldberg Y. Assessing the ability of LSTMs to learn syntax-sensitive dependencies[J]. Transactions of the Association for Computational Linguistics, 2016, 4: 521-535.