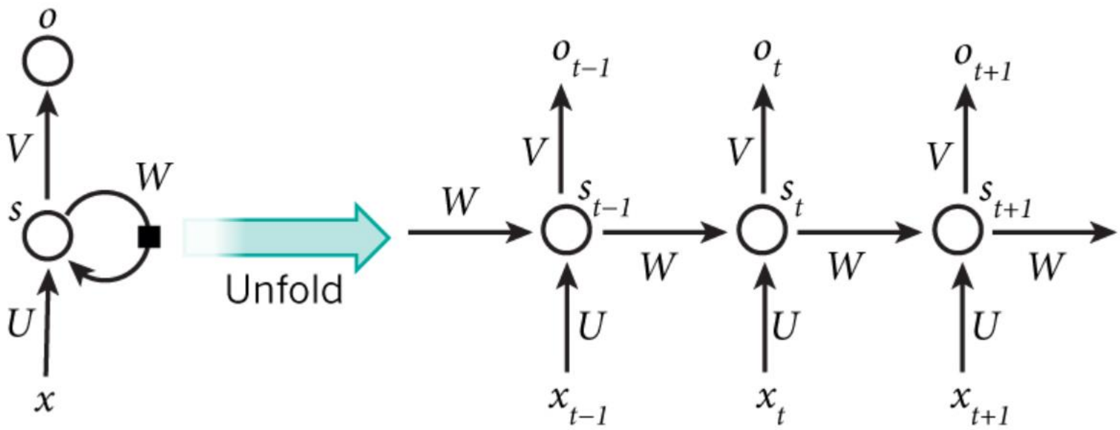
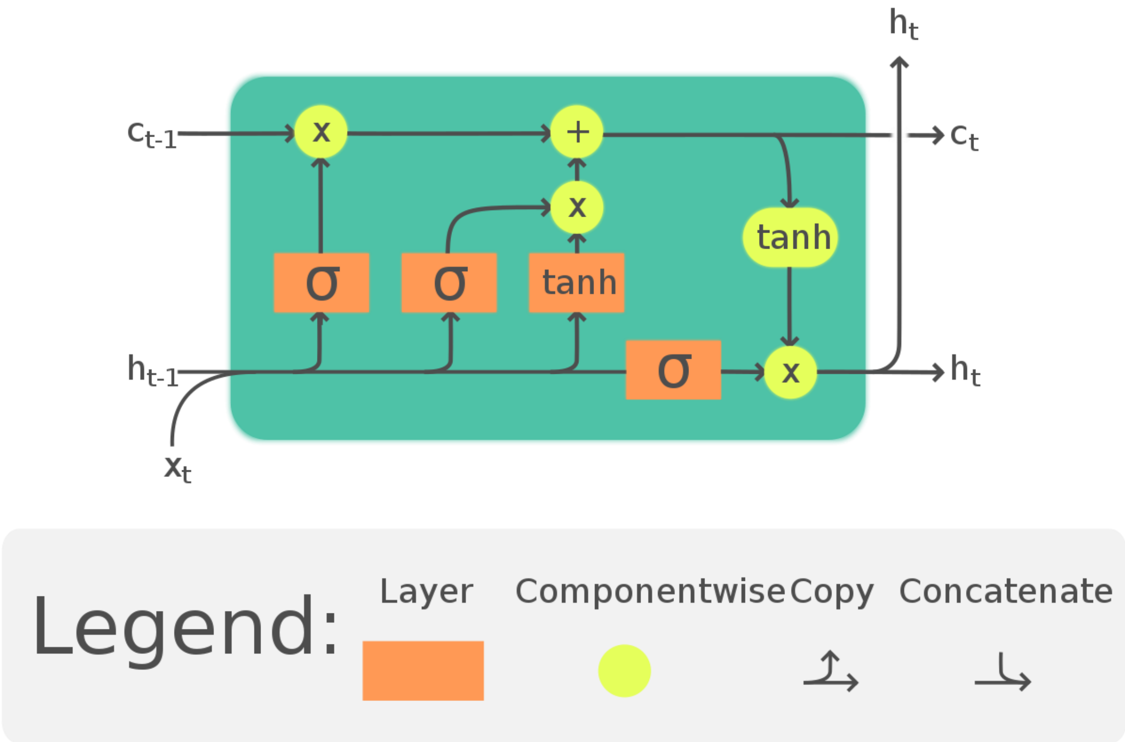
Recurrent Neural Network (RNN)



s is same as h in below equations

$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}
$$

Long Short-Term Memory (LSTM)



A common LSTM unit is composed of a **cell**, an **input gate**, an **output gate**[13] and a **forget gate**.[14] The cell remembers values over arbitrary time intervals and the three *gates* regulate the flow of information into and out of the cell.

LSTM with a forget gate [edit]

The compact forms of the equations for the forward pass of an LSTM cell with a forget gate are:[1][14]

$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
\tilde{c}_t &= \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}
$$

where the initial values are $c_0 = 0$ and $h_0 = 0$ and the operator $\circ$ denotes the Hadamard product (element-wise product). The subscript t indexes the time step.

Variables [edit]

- $x_t \in \mathbb{R}^d$: input vector to the LSTM unit
- $f_t \in (0,1)^h$: forget gate's activation vector
- $i_t \in (0,1)^h$: input/update gate's activation vector
- $o_t \in (0,1)^h$: output gate's activation vector
- $h_t \in (-1,1)^h$: hidden state vector also known as output vector of the LSTM unit
- $\tilde{c}_t \in (-1,1)^h$: cell input activation vector
- $c_t \in \mathbb{R}^h$: cell state vector
- $W \in \mathbb{R}^{h\times d}$, $U \in \mathbb{R}^{h\times h}$ and $b \in \mathbb{R}^h$: weight matrices and bias vector parameters which need to be learned during training

where the superscripts d and h refer to the number of input features and number of hidden units, respectively.

Useful Pages:

https://goodboychan.github.io/python/deep_learning/tensorflow-keras/2020/12/06/01-RNN-Many-to-one.html

https://machinelearningmastery.com/an-introduction-to-recurrent-neural-networks-and-the-math-that-powers-them/

https://machinelearningmastery.com/recurrent-neural-network-algorithms-for-deep-learning/

https://machinelearningmastery.com/understanding-simple-recurrent-neural-networks-in-keras/

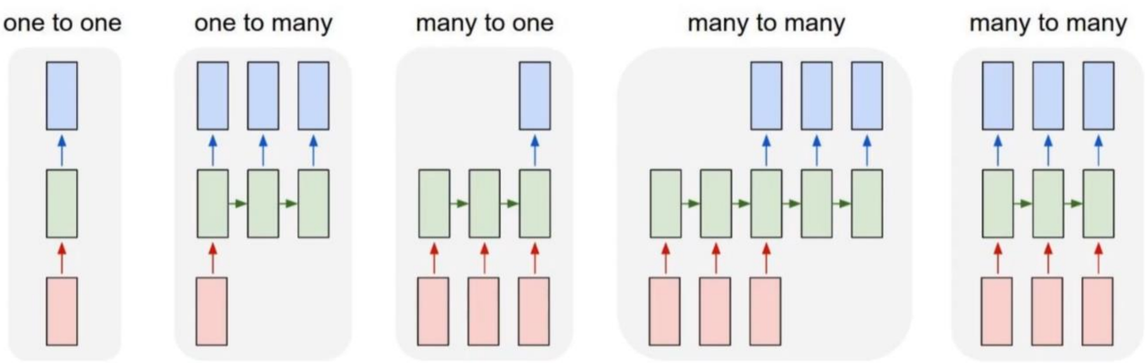
https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networks-experts/

https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/

https://www.ssla.co.uk/long-short-term-memory/

Various usage of RNN

As we already discussed, RNN is used for sequence data handling. And there are several types of RNN architecture.

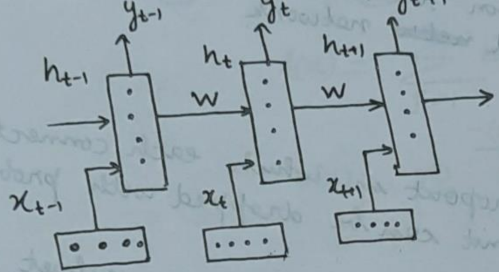


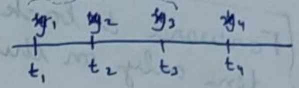
Recurrent Neural Networks

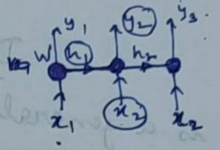
- RNNs are often used for handling sequential data
- First introduced in 1986
- Sequential data usually involves variable length inputs

Parameter sharing

Parameter sharing makes it possible to extend and apply the model to examples of different lengths and generalize across them.



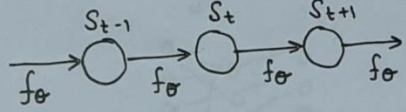




Dynamic Systems

The classical form of a dynamical system:

$$S_t = f_\theta(S_{t-1})$$



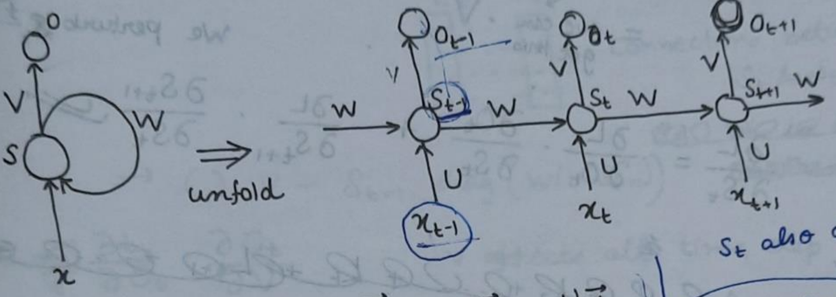
Now consider a dynamic system with an external signal x

$$S_t = f_\theta(S_{t-1}, x_t)$$

The state contains information about the whole past sequence

$$S_t = g_t(x_t, x_{t-1}, x_{t-2} \ldots, x_2, x_1)$$

- We can think of S_t as a summary of the past sequence of inputs upto t.
- If we define a different function g_t for each possible sequence length we would not get any generalization
- If same parameters are used for any sequence length allowing much better generalization properties



S_t also denoted as h_t → hidden state

softmax:-

$$P(y=j \mid \vec{x}) = \frac{e^{\vec{x}^T \vec{w}_j}}{\sum_{k=1}^{K} e^{\vec{x}^T \vec{w}_k}}$$

$$\vec{a}_t = \vec{b} + W\vec{S}_{t-1} + U\vec{x}_t$$

$$\vec{S}_t = \tanh(\vec{a}_t)$$

$$\vec{O}_t = \vec{c} + V\vec{S}_t$$

$$\vec{p}_t = \text{softmax}(\vec{O}_t)$$

Say x_t as dim = 1

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

consider h_t as dimension 3

so. no. of parameter

h_{t-1} dim = 3

No. of parameter

$W_{hh} = 3 \times 3 = 9$, $W_{xh} = 3 \times 1$

$b_t = 3$

$= 15$ para.

Computing the Gradient in a Recurrent Neural Network

Using the generalized back-propagation algorithm one can obtain the so-called Back-Propagation Through Time (BPTT) algorithm.

→ If $x_t = 1 \times 1$

$h_t = 4 \times 1$

Total parameter $= 16 + 4 + 4 = 24$

L = Total loss

L_t = Loss a t_{th} position

$$L = \sum_t L_t$$

$$\frac{\partial L}{\partial O_t} = \underbrace{\cancel{\frac{\partial L}{\partial L_t}}}_{=1} \cdot \frac{\partial L_t}{\partial O_t}$$ → will depend on loss func choosen

$L \to f(O_t, S_{t+1})$

$$\frac{\partial L}{\partial S_T} = \frac{\partial L}{\partial O_T} \cdot \frac{\partial O_T}{\partial S_T}$$

$$= \text{we can get this} \cdot V$$

$L \to$ is func. of both O_t & O_{t+1}

We perturb $S_t \xrightarrow{f} (S_{t+1})$

$$\frac{\partial L}{\partial S_t} = \frac{\partial L}{\partial O_t} \cdot \frac{\partial O_t}{\partial S_t} + \frac{\partial L}{\partial S_{t+1}} \cdot \frac{\partial S_{t+1}}{\partial S_t}$$

derivative of $\tanh(x)$

$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

$\frac{d}{dx}(\tanh(x)) = 1 - \tanh^2 x$

$$S_{t+1} = \tanh(\underbrace{W S_t + W x_t + b_t}_{M})$$

$$\frac{\partial S_{t+1}}{\partial S_t} = \frac{\partial \tanh(M)}{\partial M} \cdot \frac{\partial M}{\partial S_t} = (1 - \tanh^2(M)) \cdot W = (1 - S_t^2) W$$

$$O_t = V S_t$$

$$\frac{\partial L}{\partial S_t} =$$

Partial derivative rule :-

$L = f(O_t, S_{t+1})$

$O = f_1(S_t)$

$S_{t+1} = f_2(S_t)$

$$\frac{\partial L}{\partial S_t} =$$

e.g.

$\omega = f(x, y, z)$

$x = g(r, s)$, $y = h(r, s)$ $z = K(r, s)$

$$\frac{\partial \omega}{\partial r} = \frac{\partial \omega}{\partial x} \cdot \frac{\partial x}{\partial r} + \frac{\partial \omega}{\partial y} \cdot \frac{\partial y}{\partial r} + \frac{\partial \omega}{\partial z} \cdot \frac{\partial z}{\partial r}$$

$$\to \underbrace{y}_{\text{we can compute it}} \cdot V + \underbrace{\frac{\partial L}{\partial S_{t+1}}}_{\to \text{ again same situation } \delta_{t+1}} \cdot W \cdot (1 - S_{t+1}^2)$$

$$\vec{S}_{t+1} = \tanh(\vec{a}_{t+1})$$

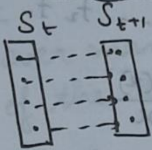
$$\vec{S}_{t+1} = \tanh(\vec{b} + W\vec{S}_t + U\vec{x}_{t+1})$$

$$\frac{\partial \vec{S}_{t+1}}{\partial \vec{S}_t} = W(1 - \tanh^2(\vec{b} - W\vec{S}_t + U\vec{x}_{t+1}))$$

$$= \underbrace{W(1 - S_{t+1}^2)}$$

→ this must be matrix but diagonal ∵ corr. elements are only connected

$$\to (\;) \cdot V + \delta_{t+1} \cdot \underbrace{W(1 - S_{t+1}^2)}$$



connections between S_t & S_{t+1}

~~W is a diagonal matrix~~

$$\to (\;) \cdot V + \delta_{t+1} \cdot \mathrm{diag}(W(1 - S_{t+1}^2))$$

$$\frac{\partial L}{\partial V} = \sum_t \frac{\partial L}{\partial O_t} \cdot \frac{\partial O_t}{\partial V}$$

V affects all time step

$$= \sum_t (\;) \cdot S_t$$

$$\frac{\partial L}{\partial W} = \sum_t \frac{\partial L}{\partial S_t} \cdot \frac{\partial S_t}{\partial W}$$

$$= \sum_t (\;) \cdot S_{t-1}(1 - S_t^2)$$

<u>Exploding or Vanishing Product of Jacobians</u>

- In recurrent nets (also in very deep nets), the final output is composition of a large number of non-linear transformations.
- Even if each of these non-linear transformations is smooth. Their composition might not be
- The derivatives through the whole composition will tend to be either very small or very large.

The Jacobian (matrix of derivatives) of a composition is the product of the Jacobians of each stage

If $f = f_T \circ f_{T-1} \circ f_{T-2} \circ \dots f_2 \circ f_1$

where $(f \circ g)(x) = f(g(x))$

$(f \circ g)'(x) = (f' \circ g)(x) \cdot g'(x) = f'(g(x)) \cdot g'(x)$

The Jacobian matrix of derivatives of $f(\vec{x})$ w.r.t. its input vector $\vec{x}$ is

$$f' = f'_T f'_{T-1} \dots f'_2 f'_1$$

where $f' = \dfrac{\partial f(x)}{\partial x}$

and $f'_t = \dfrac{\partial f_t(a_t)}{\partial a_t}$

where $a_t = f_{t-1}(f_{t-2}(\dots f_2(f_1(x))))$

Simple e.g.

Suppose : all numbers in the product are scalar and have same value α

multiplying many numbers together tends to be either very large or very small

* generally vanishing gradient happens

If T goes to ∞, then

α^T goes to ∞ if $\alpha > 1$

α^T " " 0 if $\alpha < 1$

Difficulty of Learning Long-Term dependencies

Consider a general dynamic system :

$$S_t = f_\theta(S_{t-1}, x_t)$$

$$O_t = g_w(S_t)$$

A loss L_t is computed at time step t as a func. of O_t & some target y_t. At time T:

$$\frac{\partial L_T}{\partial \theta} = \sum_{t \le T} \frac{\partial L_t}{\partial S_t} \cdot \frac{\partial S_t}{\partial \theta}$$

$$\frac{\partial L_T}{\partial \theta} = \sum_{t \le T} \frac{\partial L_t}{\partial S_T} \cdot \frac{\partial S_T}{\partial S_t} \frac{\partial f_\theta(S_{t-1}, x_t)}{\partial \theta}$$

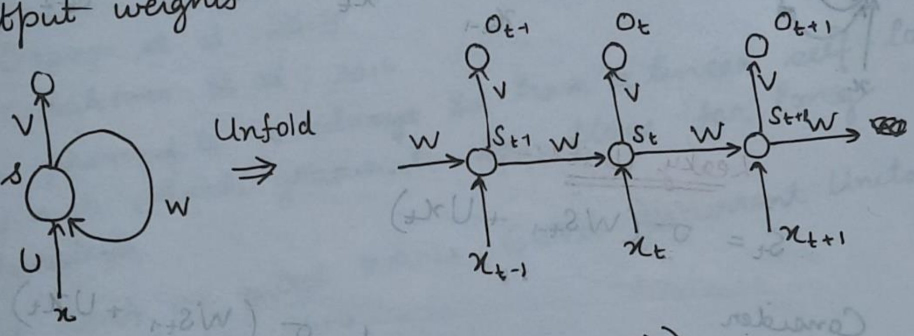
$$\frac{\partial S_T}{\partial S_t} = \frac{\partial S_T}{\partial S_{T-1}} \cdot \frac{\partial S_{T-1}}{\partial S_{T-2}} \cdots \frac{\partial S_{t+1}}{\partial S_t}$$

Facing the challenge

Gradients propagated over many stages tend to either vanish (most of the time) or explode.

Echo State Networks

Set the recurrent and input weights such that the recurrent hidden units do a good job of capturing the history of past inputs and only learn the output weights



$$S_t = \sigma(W\vec{S}_{t-1} + U\vec{x}_t)$$

→ W is set in a way that Jacobian is stable i.e. eigenvalues of the Jacobian is little less than 1 (i.e. eventually forgets)
It will vanish in long time. so no problem

→ One idea is to set $W = I$ & learn V
& even $U = I$

– ReLU is used

If a change Δs in the state at t is aligned with an eigenvector v of Jacobian J with eigenvalue $\lambda > 1$, then the small change Δs becomes $\lambda \Delta s$ after one-time step, and $\lambda^t \Delta s$ after t time steps

If the largest eigenvalue $\lambda < 1$, the map from t to $t+1$ is contractive

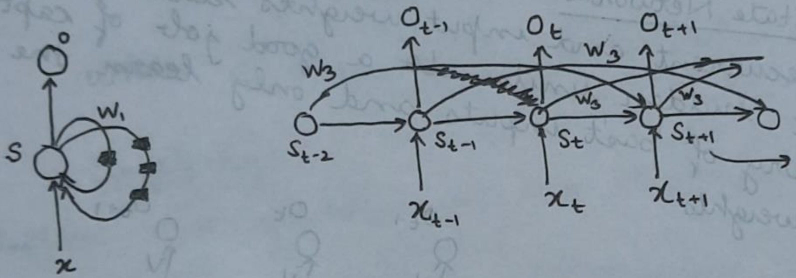
The network forgetting information about the long-term past

Set the weights to make Jacobians slightly contractive

Another way of handling vanishing gradients is

Long Delays :-

Use recurrent connections with long delays :-



S_{t+1} is affected by S_{t-2} through W_3

Leaky Units:-

$$S_t = \sigma(W S_{t-1} + U x_t)$$

Consider

$$S_{t,i} = \left(1 - \frac{1}{\tau_i}\right) S_{t-1} + \frac{1}{\tau_i} \sigma(W S_{t-1} + U x_t)$$

ith component of state vector

$$1 \leq \tau_i \leq \infty$$

$\tau_i = 1$, Ordinary RNN

$\tau_i > 1$, gradients propagate more easily

$\tau_i >> 1$, the state changes very slowly, integrating the past values associated with input sequence

Gated RNNs

It might be useful for the neural network to forget the old state in some cases:-

Example: a a b b b a a a a b a b

It might be useful to keep the memory of the past

Example:

Instead of manually deciding when to clear the state, we want the neural network to learn to decide when to do it.

Gated RNNs, the Long-Short-Term-Memory

The Long-Short-Term-Memory (LSTM) algorithm was proposed in 1997 (Hochreiter & Schmidhuber, 1997)

Several variants of the LSTM are found in literature:-

Hochreiter & Schmidhuber 1997

Graves, 2012

Graves et al., 2013

Sutskever et al., 2014

the principle is always to have a linear self-loop through which gradients can flow for long duration.

Recent work on gated RNNs, Gated Recurrent Units (GRU) was proposed in 2014

Cho et al., 2014

Chung et al., 2014, 2015

Jozefowicz et al., 2015

Chrupala et al., 2015

→ GRU less flexible in LSTM

Gated Recurrent Units (GRU)

Standard RNN computes hidden layer at next time step directly:

$\sigma \rightarrow$ sigmoid func.

$$h_t = f(Wh_{t-1} + Ux_t)$$

GRU first computes an update Gate (another layer) based on current input vector and hidden state

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1})$$

compute reset gate similarly with different weights

$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1})$$

Update gate: $z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1})$

Reset gate: $r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1})$

New memory content: $\tilde{h}_t = \tanh(Wx_t + r_t \circ Uh_{t-1})$

If reset gate is 0, then ignores previous memory and only stores the new information

Final memory at time step combines current and previous time steps:-

$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$$

The Long Short-Term Memory (LSTM)

We can make the units even more complex

Allow each time step to modify

Input gate (current cell matters) $i_t = \sigma(W^{(i)}x_t + U^{(i)}h_{t-1})$

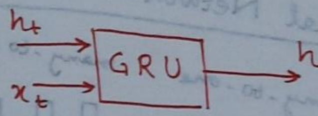
Forget (gate 0, forget past) $f_t = \sigma(W^{(f)}x_t + U^{(f)}h_{t-1})$

Output (how much cell is exposed) $o_t = \sigma(W^{(o)}x_t + U^{(o)}h_{t-1})$

New memory cell $\tilde{c}_t = \tanh(W^{(c)}x_t + U^{(c)}h_{t-1})$

Final memory cell, $c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$

Final hidden state: $h_t = o_t \circ \tanh(c_t)$



→ If reset is close to 0, ignore previous hidden state → Allow model to drop information that is irrelevant

update gate z controls how much of past state should matter now. If z close to 1, then we copy information in that unit through many time steps.

Units with short term dependencies often have reset gates very active

Clipping Gradients

Strongly non-linear functions tend to have derivatives that can be either very large or very small in magnitude

- If gradient is greater than some threshold (say 2) clip it to 2.

Possible Solutions :-

- Simple sol. for clipping gradient from a ~~β~~ mini batch
- Clip the parameter gradient from a the parameter update element-wise just before the parameter update
- Clip the norm g of gradient g just before the parameter update.