Convolutional and Recurrent Neural Networks

ML4SE

Denis Litvinov

November 9, 2021

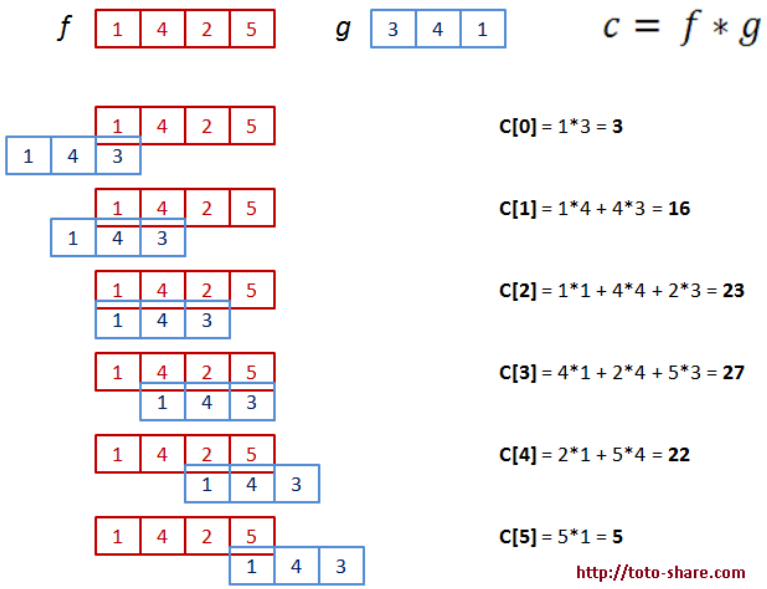
Table of Contents

Quiz

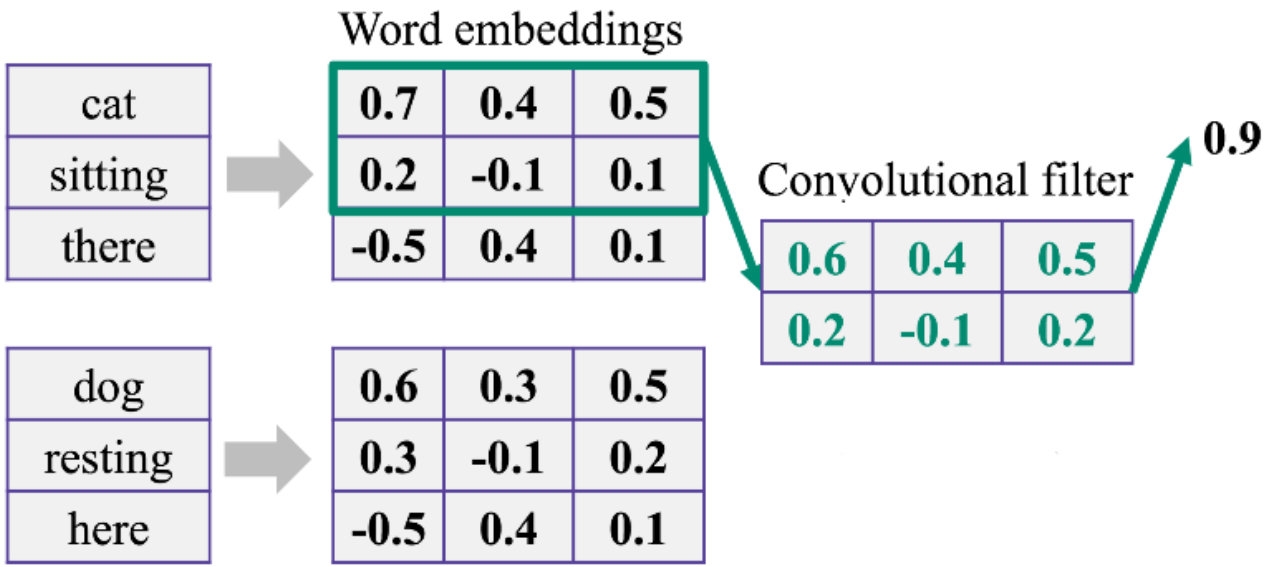
Convolution

Formal definition of convolution of functions f and g

$$(fg)(t) = \sum_{j=1}^{m} g(j) f(i - j + m/2)$$

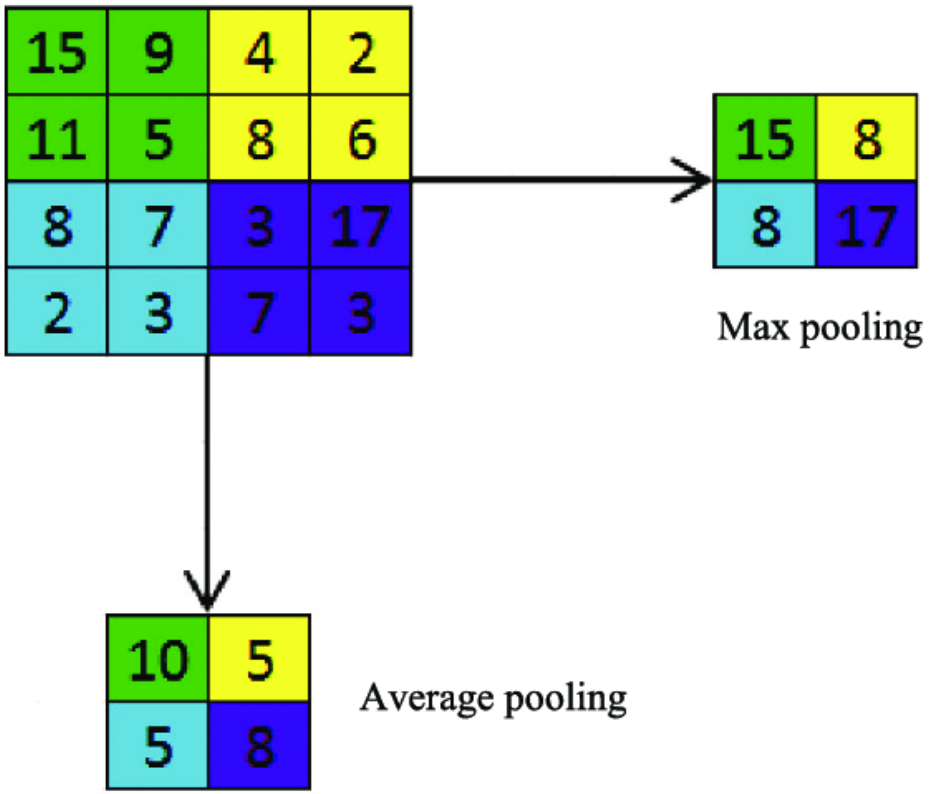


Convolution



- Is convolution a linear operation?

Pooling

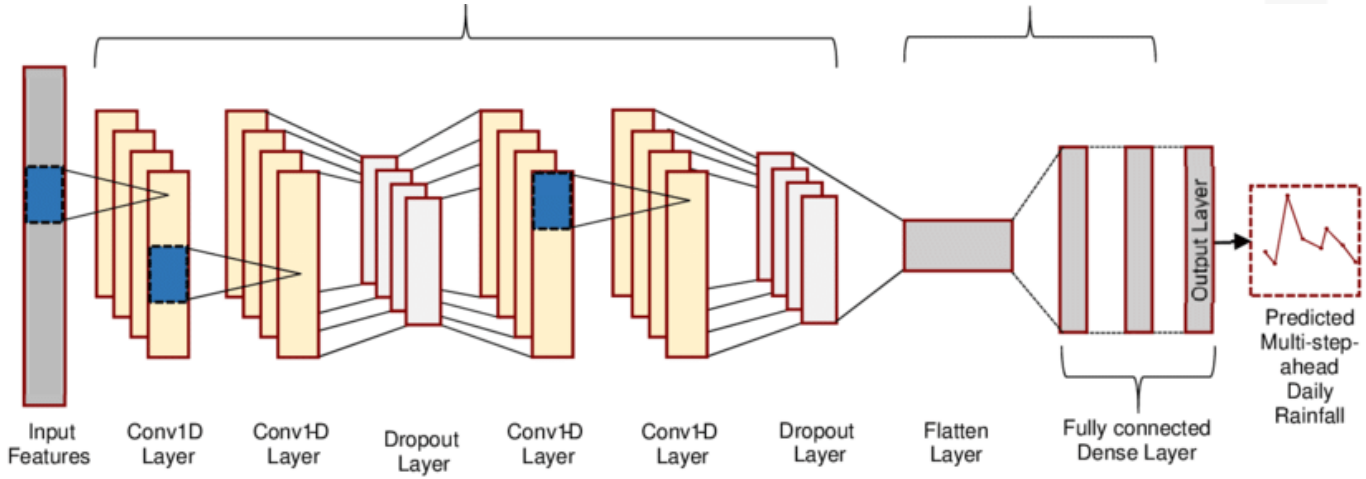

15 9 4 2
11 5 8 6
8 7 3 17
2 3 7 3
15 8
8 17
Max pooling
10 5
5 8
Average pooling

Common Architectures

- Conv
- BatchNorm
- Activation
- Pooling
- Dropout

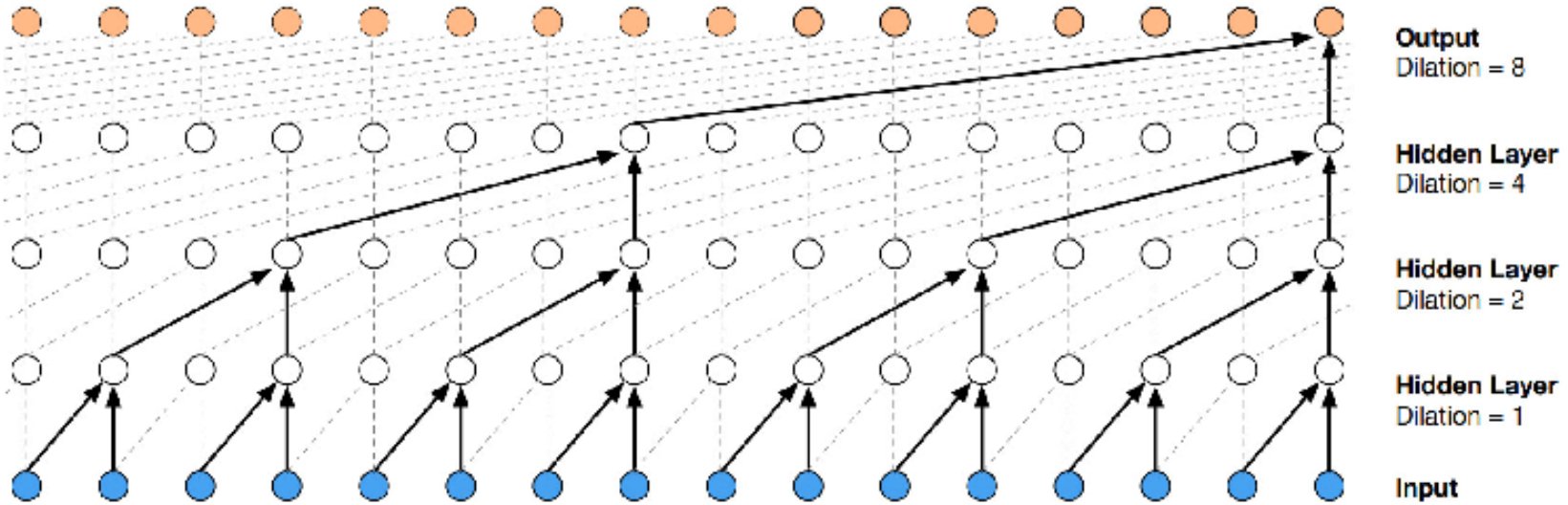
(omit if necessary)

Stacked Convolutions = Place Conv blocks on top of each other



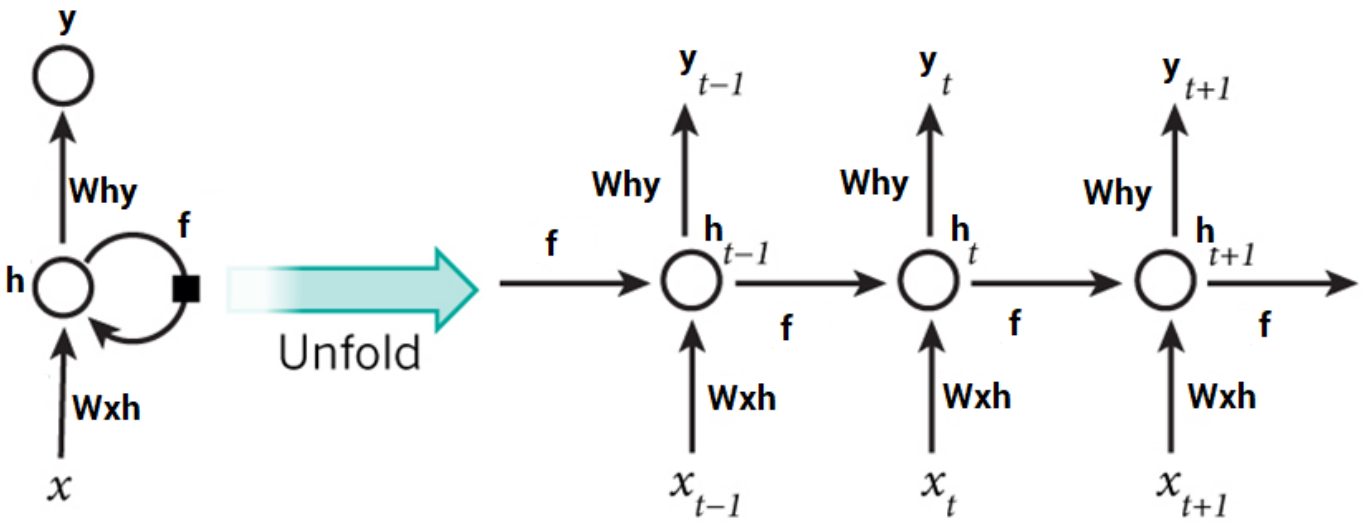
Wide convolutions = Place several Conv blocks in parallel with different kernel size and concatenate the output

Dilated Convolution



Demo

Vanilla Recurrent Unit



In general, can be described by 2 equations:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta)$$

$$o^{(t)} = g(h^{(t-1)}, \theta)$$

Vanilla Recurrent Unit

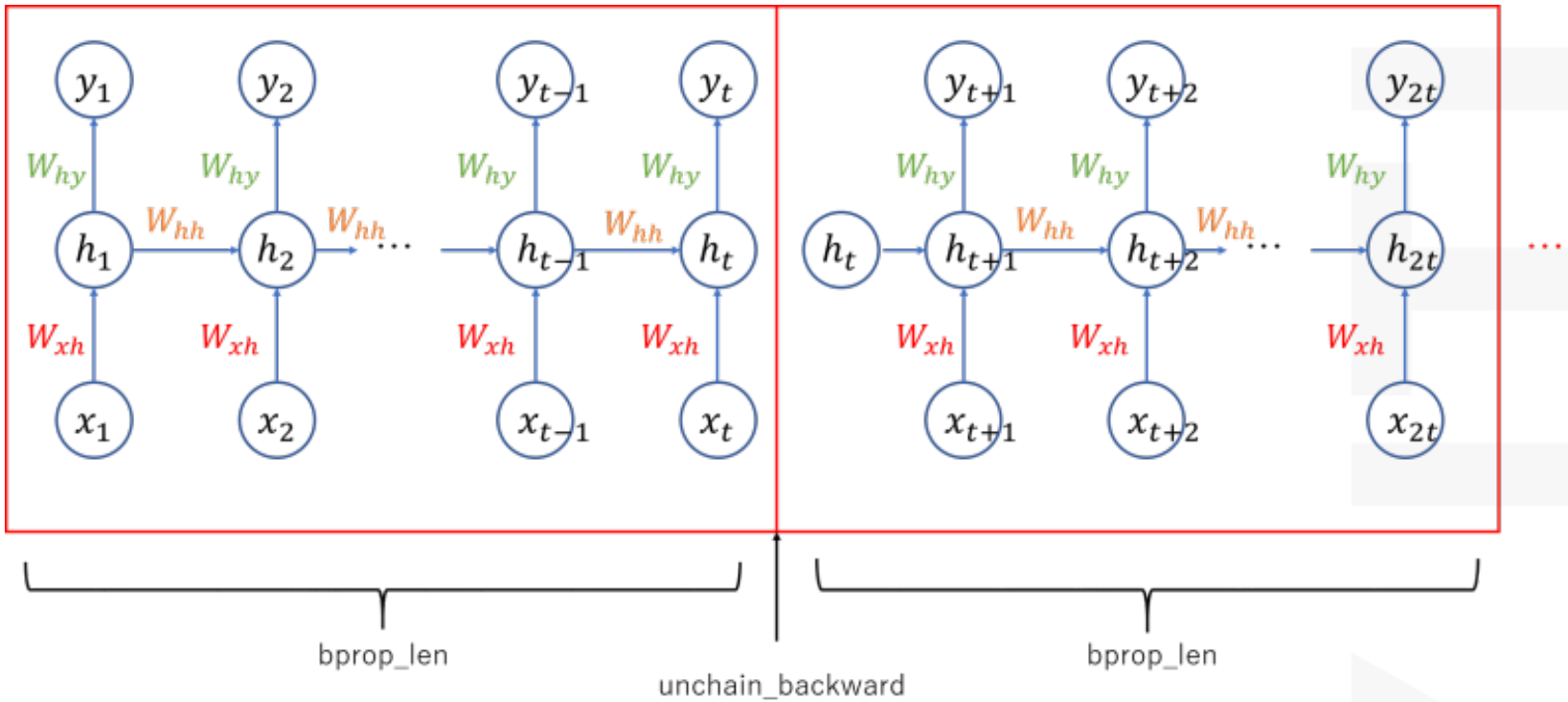
where $x^{(t)}$ - input at time t,
$h^{(t)}$ - hidden state at time t,
$o^{(t)}$ - output at time t,
f, g - some functions, parametrized by learnable weights θ
For Vanilla RNN usually:

$$h^{(t)} = \tanh(b_h + W_{hh}h^{(t-1)} + W_{hx}x^{(t)})$$

$$o^{(t)} = softmax(b_o + W_{oh}h^{(t)})$$

If not specified, $h^{(0)}$ is initialized with zeros.

Backpropagation Through Time



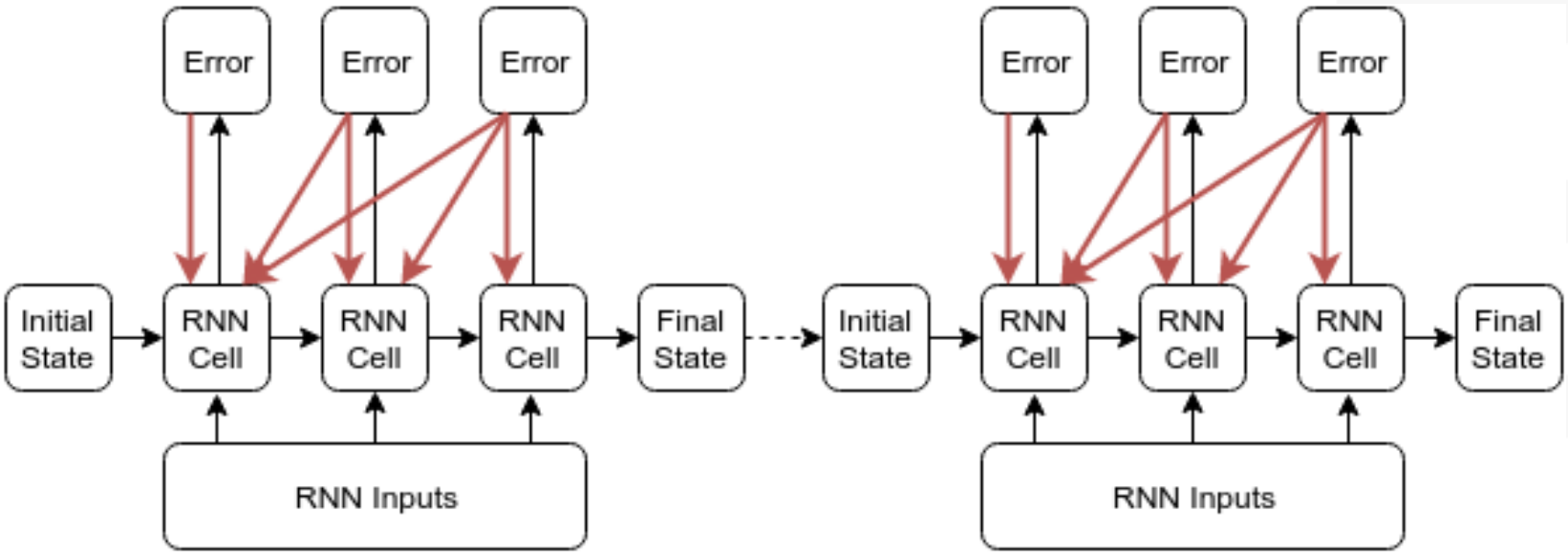
Backpropagation Through Time

Because every next step of RNN becomes dependent on the previous step, it means that gradient estimation becomes dependent on the length of the sequence. Weights of RNN are shared across steps and gradients from every time step will be summed.
For example, for $T = 3$:

$$\frac{\partial L}{\partial W_{hh}} = \frac{\partial L}{\partial h^{(3)}} \frac{\partial h^{(3)}}{\partial W_{hh}} + \frac{\partial L}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial W_{hh}} + \frac{\partial L}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial W_{hh}}$$

To make gradient estimation more manageable, in BPTT we introduce a sliding window, where we compute gradients and make weight updates.
Though, these complexities are usually covered by your deep learning framework

Backpropagation Through Time



Problems with Vanilla RNN

Vanilla RNNs are subject to exploding and vanishing gradients, which makes training unstable.
Look closely on

$$h^{(t)} = \tanh(b_h + W_{hh}h^{(t-1)} + W_{hx}x^{(t)})$$

Let's try to compute $\frac{\partial L}{\partial h}$.
For the sake of simplicity, lets change activation function *tanh* $\rightarrow 1$, which make sense, if we make the approximation around 0, where *tanh* has linear behavior.
Also, suppose that only the last time step is used for loss computation, so L only depends on $h^{(N)}$.

Problems with Vanilla RNN

So,

$$h^{(t)} = b_h + W_{hh}h^{(t-1)} + W_{hx}x^{(t)}$$

$$\frac{\partial L}{\partial h} = \frac{\partial L}{\partial h^{(N)}}(W_{hh}\frac{\partial h^{(N)}}{\partial h^{(N-1)}})(W_{hh}\frac{\partial h^{(N-1)}}{\partial h^{(N-2)}})...(W_{hh}\frac{\partial h^{(1)}}{\partial h^{(0)}}) = \frac{\partial L}{\partial h^{(N)}}W_{hh}^N$$

Because W_{hh}^N is a power of square matrix, we can calculate it by eigendecomposition:

$$W_{hh}^N = U\Lambda^N U^T$$

, where once again, U - unitary matrix, and Λ - diagonal matrix
Now,

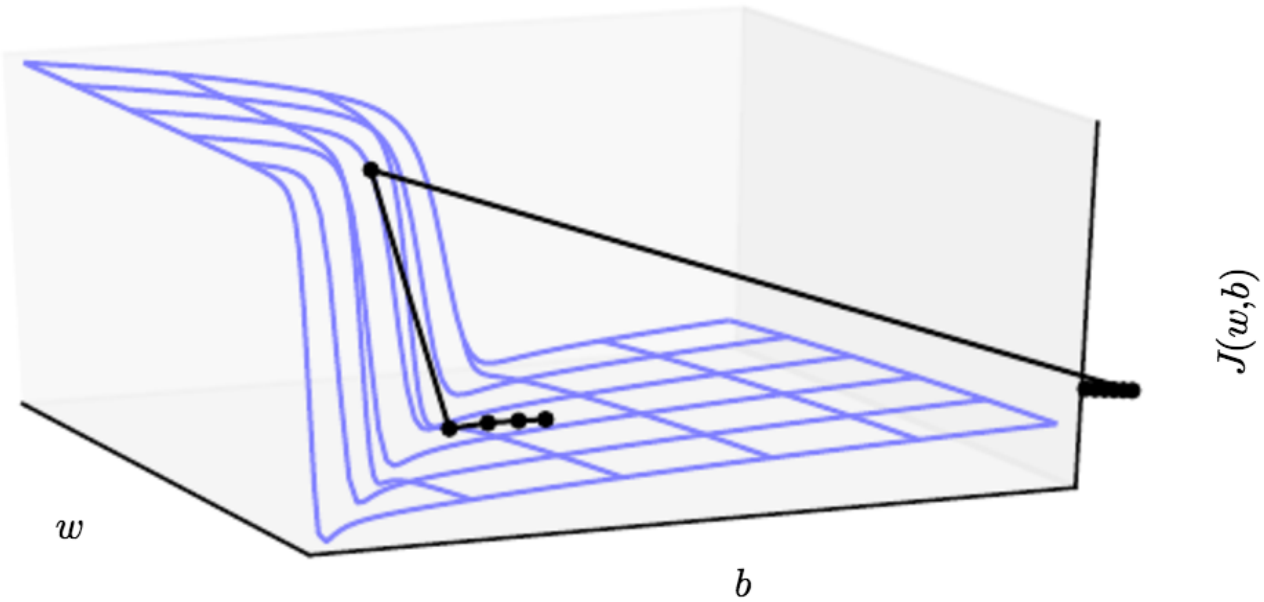
$$\frac{\partial L}{\partial h} = \frac{\partial L}{\partial h^{(N)}}U\Lambda^N U^T$$

Problems with Vanilla RNN

Our gradient becomes highly dependent on the conditional number (relation between maximum and minimum eigenvalue) $cond(W_{hh}) = \frac{\max(\lambda_i i)}{\min(\lambda_i i)}$

For example, if we have $cond(W_{hh}) > 1$, then we get exploding gradients. Because

$$\lim_{N \to \infty} 1.001^N = \infty$$



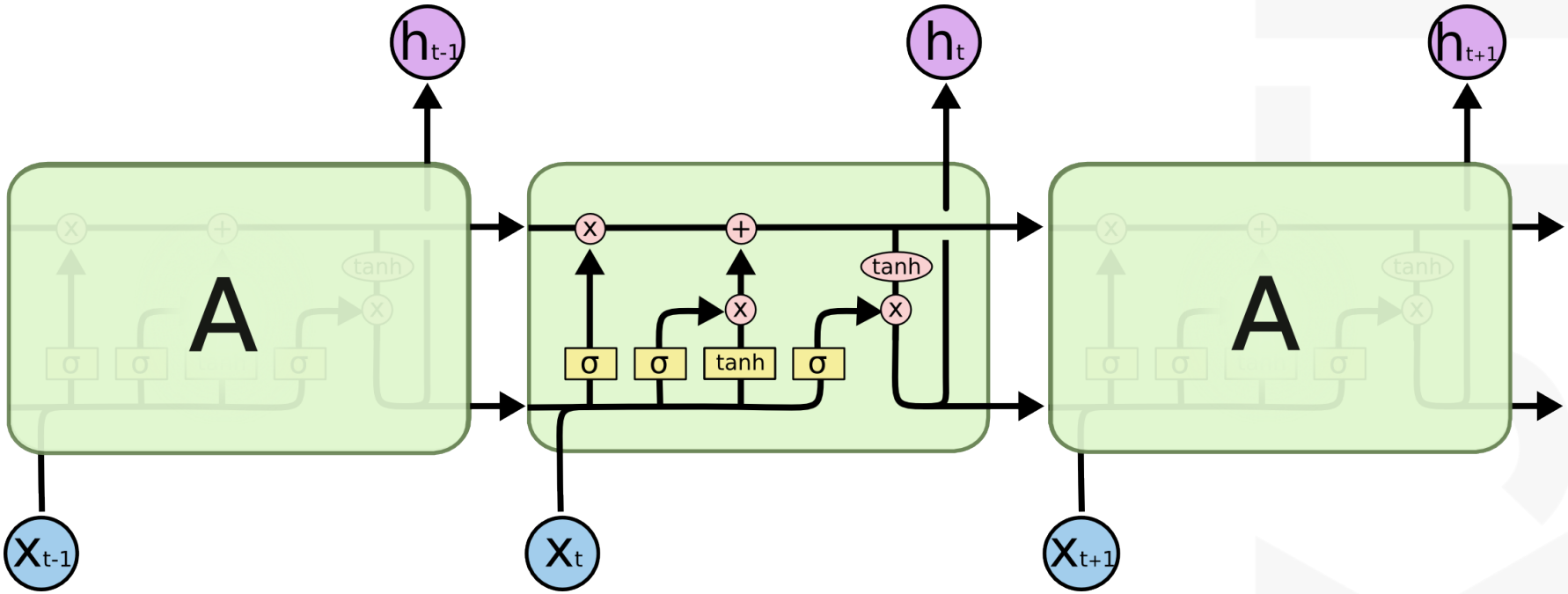
Problems with Vanilla RNN

Problem with exploding gradient is that we can move too far from our optimal solution. Exploding gradients can be managed by gradient clipping.
If we have $cond(W_{hh}) < 1$, then we have vanishing gradients. Because

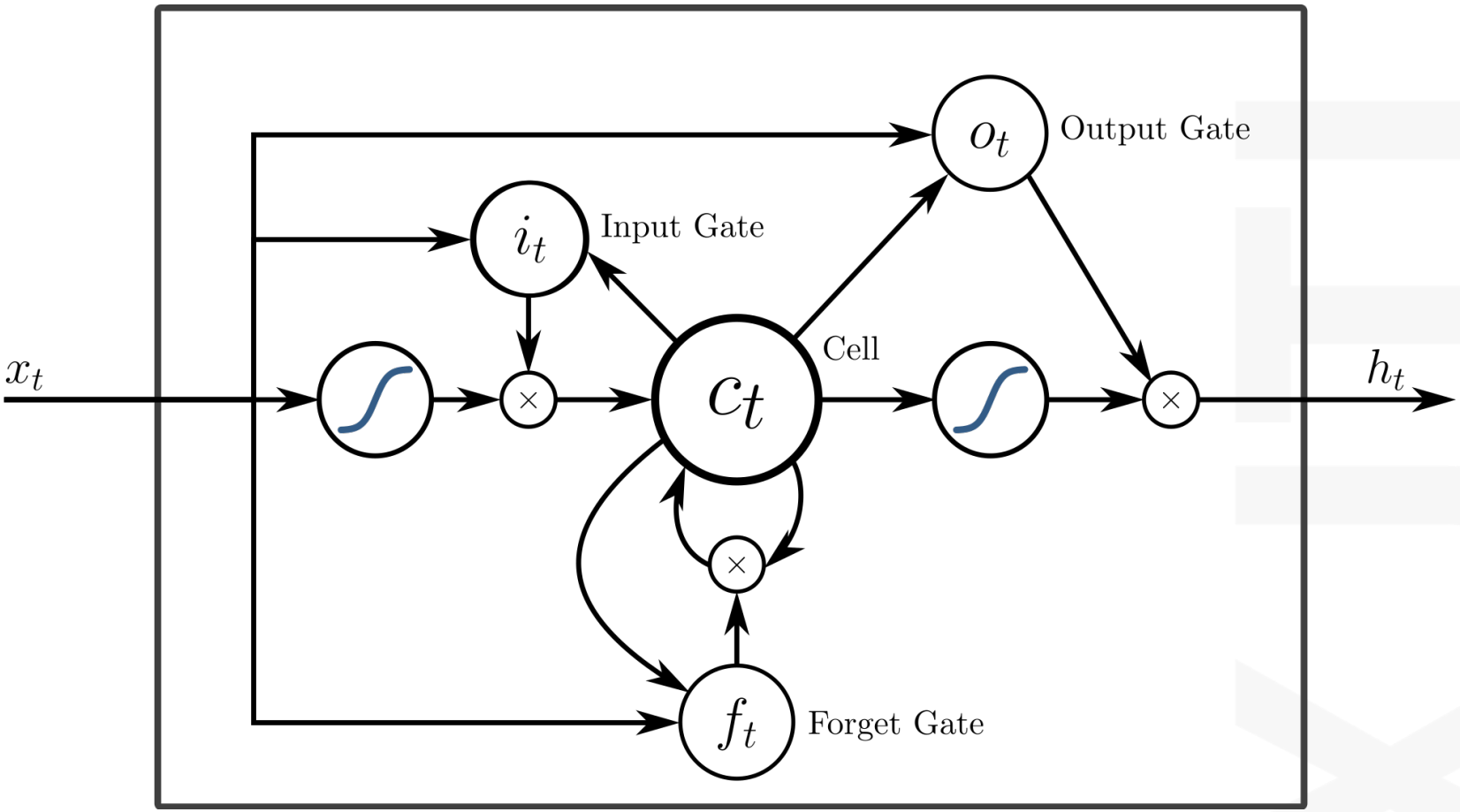
$$\lim_{N \to \infty} 0.999^N = 0$$

Problem with vanishing gradient is that we can think, that the optimal solution is found. Remember, that in a local minimum the gradient is also zero.

Long Short-Term Memory



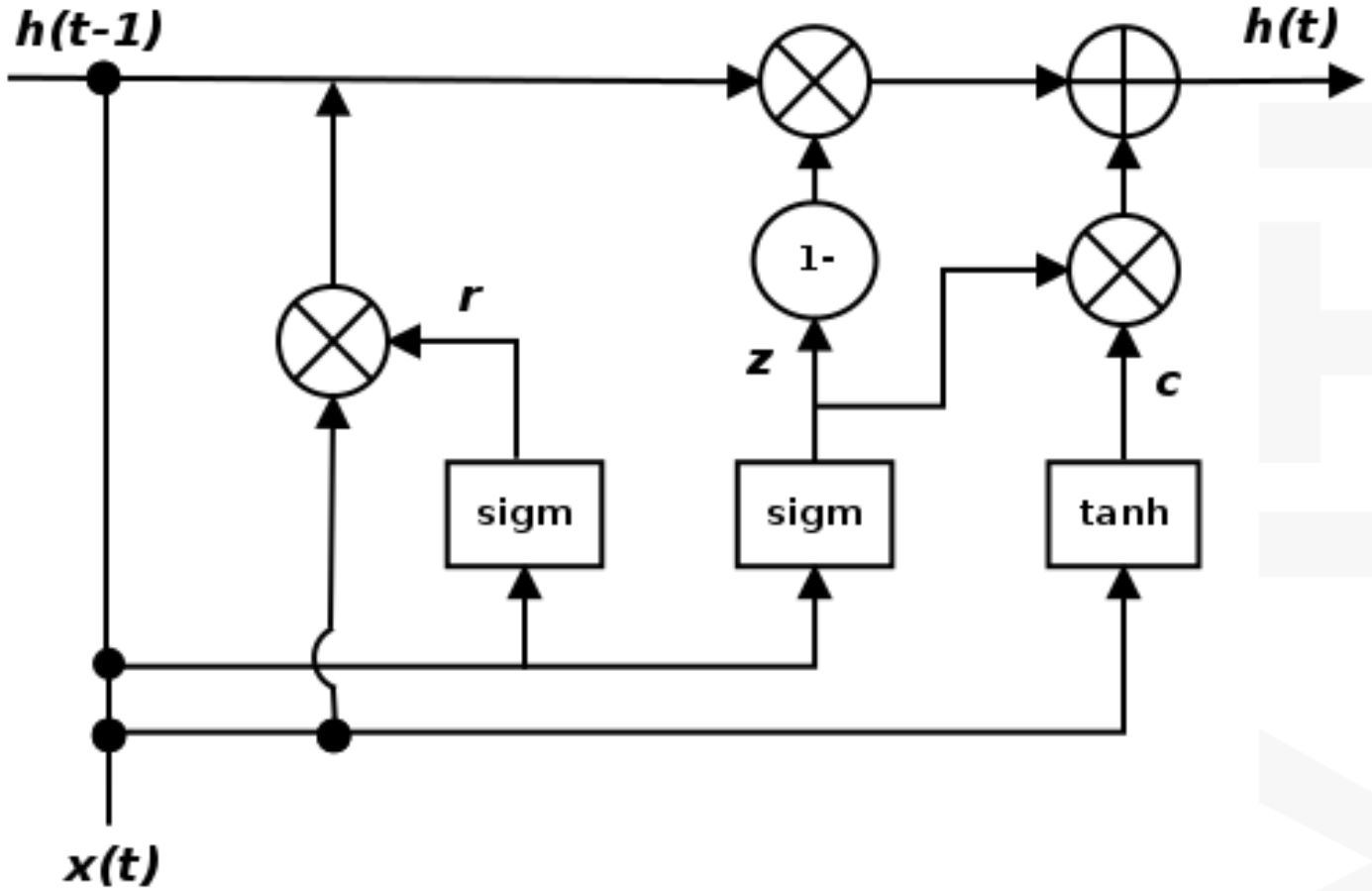
Long Short-Term Memory



Long Short-Term Memory

f_t - forget gate,
i_t - input gate,
o_t - output gate,
c_t - cell memory,
h_t - hidden state
Compared to Vanilla RNN, cell state in LSTM allows "free" gradient propagation without reccurent relations.
There are 4 times more parameters in LSTM cell, then in RNN cell.

Gated Recurrent Unit



Gated Recurrent Unit

$$\mathbf{r}^t = \sigma(\mathbf{W}_r^d \mathbf{x}^{t-1} + \mathbf{U}_r^d \mathbf{h}^{t-1} + \mathbf{C}_r \mathbf{h}_i) \quad (5)$$

$$\mathbf{z}^t = \sigma(\mathbf{W}_z^d \mathbf{x}^{t-1} + \mathbf{U}_z^d \mathbf{h}^{t-1} + \mathbf{C}_z \mathbf{h}_i) \quad (6)$$

$$\bar{\mathbf{h}}^t = \tanh(\mathbf{W}^d \mathbf{x}^{t-1} + \mathbf{U}^d (\mathbf{r}^t \odot \mathbf{h}^{t-1}) + \mathbf{C}\mathbf{h}_i) \quad (7)$$

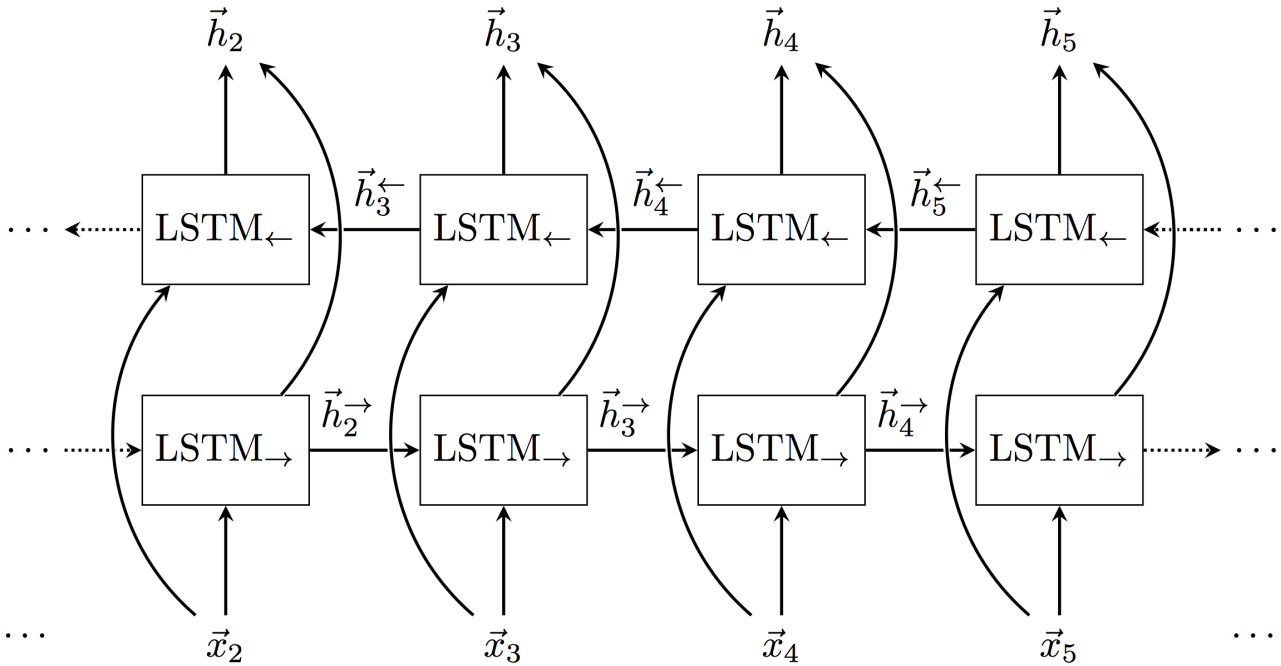
$$\mathbf{h}_{i+1}^t = (1 - \mathbf{z}^t) \odot \mathbf{h}^{t-1} + \mathbf{z}^t \odot \bar{\mathbf{h}}^t \quad (8)$$

r_t - record gate, z_t - forget gate, h_t - hidden state

- Combines input gate and forget gate
- Merges cell state and hidden state

There are 3 times more parameters in GRU cell, then in RNN cell.

Bidirectional RNN



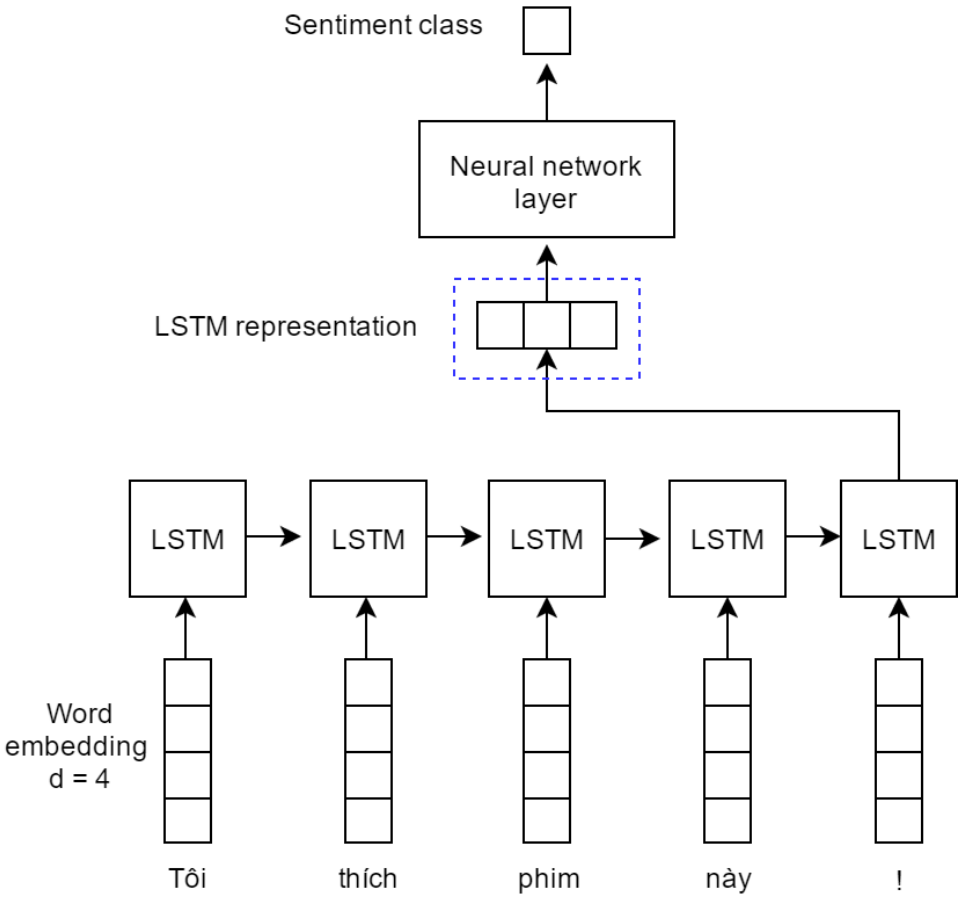
Notes

- There are 2 implementations of LSTM in pytorch: ‘torch.nn.LSTM‘ and ‘torch.nn.LSTMCell‘. ‘torch.nn.LSTM‘ is CuDNN optimized version, where all training and BPTT are handled by the library. It also allows you to make bidirectional, stack layers, or add dropout between layers. But you won’t have access to intermidiate hidden and output states. ‘torch.nn.LSTMCell‘ is a basic version of LSTM, all training and BPTT must be managed by you. Same holds for GRU.
- Though, in theory LSTM(GRU) can manage very long sequences, in practice it is not so good. Direction of RNN matters, in general last time steps will have more influence on the result. Consider Bidirectional if possible.

Notes

- By default, sequence padding is handled by you. The library will not recognize padded symbols, and will process them as if they are part of the data. It may affect your model performance and quality. Consider special packing routines to specify length of each sequence.
- Usually, RNN is more computationally expensive then CNN, because it cannot be paralleled.

RNN for classification



Demo