Lecture 5: Learning Sequential Patterns with RNN & LSTM

Assoc Prof Tham Chen Khong
Dept of Electrical & Computer Engineering (ECE)
NUS
E-mail: eletck@nus.edu.sg

Acknowledgement: some materials from Géron, Hands On ML

Overview

- Recurrent Neural Network (RNN)
- Long Short Term Memory (LSTM)
- Gated Recurrent Unit (GRU)

Introduction

- Here, we consider neural networks that can process
  - sequences / sequential data
  - time series
- and predict the next outcomes

- Examples: speech, stock market, trajectory of moving objects, language translation (e.g. Google Translate)

Recurrent Neurons and Layers

- Input and target (or output) values are presented to the network at each time step
- There are internal connections from one time step to the next time step

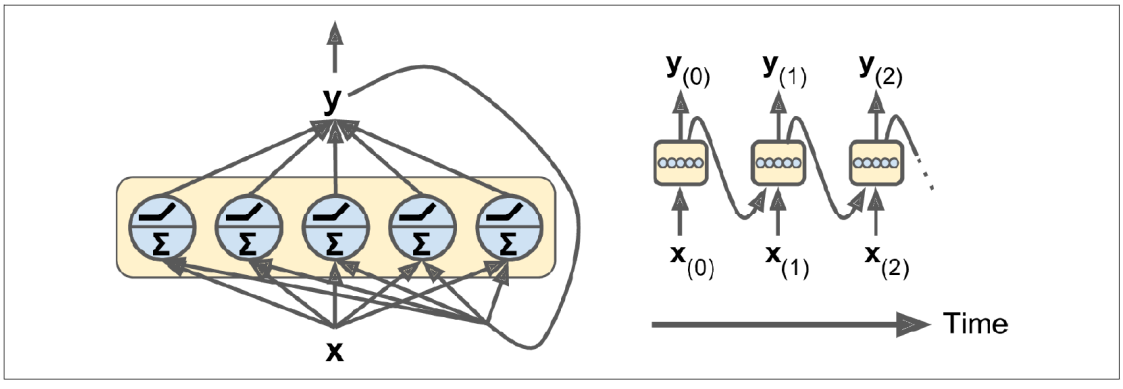


Figure 15-2. A layer of recurrent neurons (left) unrolled through time (right)

RNN

Each recurrent neuron has two sets of weights: one for the inputs $\mathbf{x}_{(t)}$ and the other for the outputs of the previous time step, $\mathbf{y}_{(t-1)}$. Let's call these weight vectors $\mathbf{w}_x$ and $\mathbf{w}_y$. If we consider the whole recurrent layer instead of just one recurrent neuron, we can place all the weight vectors in two weight matrices, $\mathbf{W}_x$ and $\mathbf{W}_y$. The output vector of the whole recurrent layer can then be computed pretty much as you might expect, as shown in Equation 15-1 ($\mathbf{b}$ is the bias vector and $\phi(\cdot)$ is the activation function (e.g., ReLU[1]).

Equation 15-1. Output of a recurrent layer for a single instance

$$\mathbf{y}_{(t)} = \phi\left(\mathbf{W}_x^{\top}\mathbf{x}_{(t)} + \mathbf{W}_y^{\top}\mathbf{y}_{(t-1)} + \mathbf{b}\right)$$

Equation 15-2. Outputs of a layer of recurrent neurons for all instances in a mini-batch

$$\mathbf{Y}_{(t)} = \phi\left(\mathbf{X}_{(t)}\mathbf{W}_x + \mathbf{Y}_{(t-1)}\mathbf{W}_y + \mathbf{b}\right)$$

$$= \phi\left(\begin{bmatrix}\mathbf{X}_{(t)} & \mathbf{Y}_{(t-1)}\end{bmatrix}\mathbf{W} + \mathbf{b}\right) \text{ with } \mathbf{W} = \begin{bmatrix}\mathbf{W}_x \\ \mathbf{W}_y\end{bmatrix}$$

RNN

In this equation:

- $\mathbf{Y}_{(t)}$ is an $m \times n_{\text{neurons}}$ matrix containing the layer's outputs at time step t for each instance in the mini-batch (m is the number of instances in the mini-batch and n_{neurons} is the number of neurons).
- $\mathbf{X}_{(t)}$ is an $m \times n_{\text{inputs}}$ matrix containing the inputs for all instances (n_{inputs} is the number of input features).
- $\mathbf{W}_x$ is an $n_{\text{inputs}} \times n_{\text{neurons}}$ matrix containing the connection weights for the inputs of the current time step.
- $\mathbf{W}_y$ is an $n_{\text{neurons}} \times n_{\text{neurons}}$ matrix containing the connection weights for the outputs of the previous time step.
- $\mathbf{b}$ is a vector of size n_{neurons} containing each neuron's bias term.
- The weight matrices $\mathbf{W}_x$ and $\mathbf{W}_y$ are often concatenated vertically into a single weight matrix $\mathbf{W}$ of shape $(n_{\text{inputs}} + n_{\text{neurons}}) \times n_{\text{neurons}}$ (see the second line of Equation 15-2).
- The notation $[\mathbf{X}_{(t)}\ \mathbf{Y}_{(t-1)}]$ represents the horizontal concatenation of the matrices $\mathbf{X}_{(t)}$ and $\mathbf{Y}_{(t-1)}$.

Notice that $\mathbf{Y}_{(t)}$ is a function of $\mathbf{X}_{(t)}$ and $\mathbf{Y}_{(t-1)}$, which is a function of $\mathbf{X}_{(t-1)}$ and $\mathbf{Y}_{(t-2)}$, which is a function of $\mathbf{X}_{(t-2)}$ and $\mathbf{Y}_{(t-3)}$, and so on. This makes $\mathbf{Y}_{(t)}$ a function of all the inputs since time $t = 0$ (that is, $\mathbf{X}_{(0)}, \mathbf{X}_{(1)}, \ldots, \mathbf{X}_{(t)}$). At the first time step, $t = 0$, there are no previous outputs, so they are typically assumed to be all zeros.

Memory Cells

- Since the output of a recurrent neuron at time step *t* is a function of all the inputs from previous time steps, it has a form of *memory*. A part of a neural network that preserves some **state** across time steps is called a *memory cell* (or simply a *cell*).

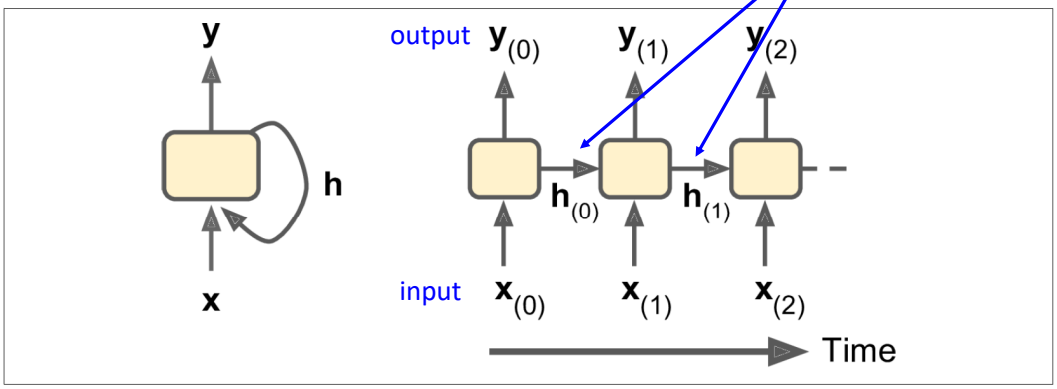


Figure 15-3. A cell's hidden state and its output may be different

- This is a simple type of cell which can only remember a few time steps. Later, we consider more powerful types of cells!

Input and Output Sequences

- *Seq-to-seq*
  - *useful for time series prediction, e.g. output is predicted future values*
- *Seq-to-vector*
  - *useful for e.g. sentiment analysis and time series prediction*
- *Vector-to-seq*
  - *feed same input over a few time steps, e.g. trigger output sequence*
- *Encoder–Decoder networks*
  - *seq-to-vector, followed by vector-to-seq*
  - *useful for, e.g. language translation*

note: Y is a vector

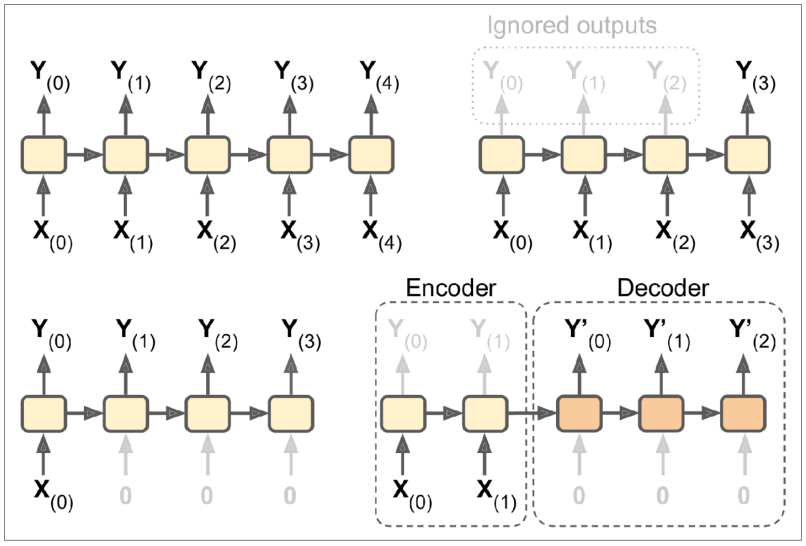


Figure 15-4. Seq-to-seq (top left), seq-to-vector (top right), vector-to-seq (bottom left), and Encoder–Decoder (bottom right) networks

Training – Back Propagation Through Time (BPTT)

Williams & Zipser, 1985
Robinson & Fallside, 1987
Werbos, 1988

seq-to-seq case

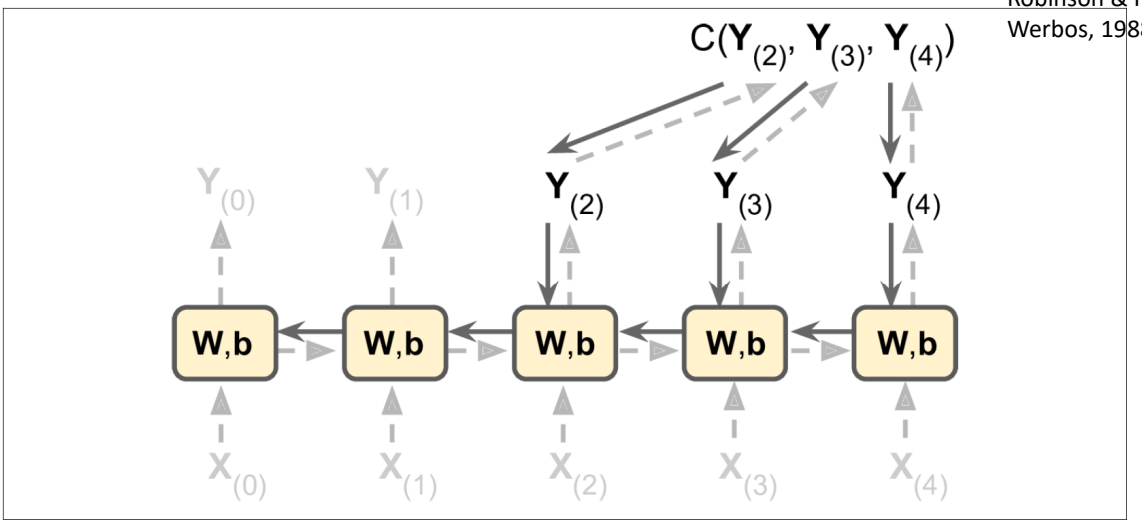


Figure 15-5. Backpropagation through time

Note: the cost function is computed using the last three outputs of the network, $\mathbf{Y}_{(2)}$, $\mathbf{Y}_{(3)}$, and $\mathbf{Y}_{(4)}$, so gradients flow through these three outputs

Training - BPTT

- Unroll the NN through time, then do normal back propagation
  → back propagation through time (BPTT)
- There is first a forward pass through the unrolled network (represented by the dashed arrows)
- Output sequence is evaluated using a cost function $C(\mathbf{Y}_{(0)}, \mathbf{Y}_{(1)}, \ldots \mathbf{Y}_{(T)})$
- Gradients of that cost function are then propagated backward through the unrolled network (represented by the solid arrows)
- Finally, the model parameters are updated using the gradients computed during BPTT.
- *Note:* the gradients flow backward through all the outputs used by the cost function, not just through the final output
- Since the same parameters **W** and **b** are used at each time step, backpropagation needs to correctly sum/accumulate weight changes over all time steps.

Deep RNN

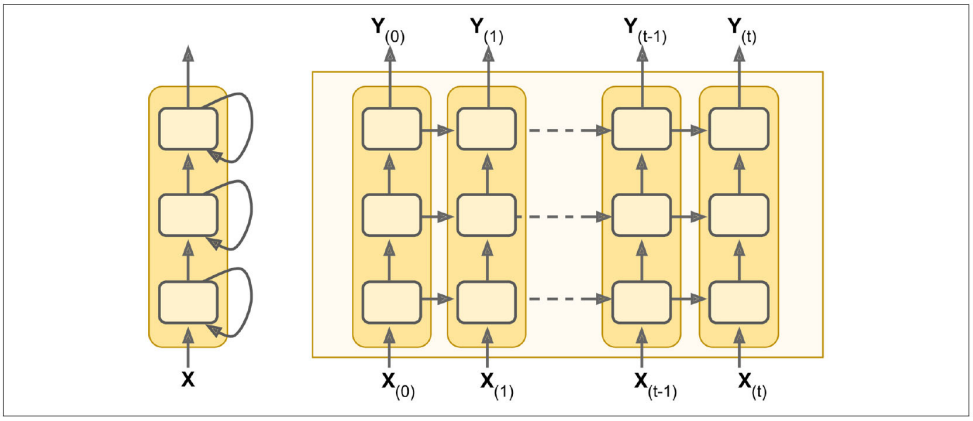


Figure 15-7. Deep RNN (left) unrolled through time (right)

Programming Tip

When dealing with time series (and other types of sequences such as sentences), the input features are generally represented as 3D arrays of shape [*batch size*, *time steps*, *dimensionality*], where *dimensionality* is 1 for univariate time series and more for multivariate time series.

Handling Long Sequences

- RNNs suffer from vanishing gradients caused by long series of multiplications of small values, diminishing the gradients and causing the learning process to become degenerate.
- Similarly, RNNs can suffer from exploding gradients arising from large gradient values which hampers the learning process.
- Batch normalization (BN) should only be used between layers and not across time steps in RNNs as as it changes the characteristics of the sequential data presented in an RNN. It may also cause unstable gradients.
- Hence, fight the unstable / vanishing gradients problem through
  - Layer normalization
  - Dropout

Layer Normalization

- Another form of normalization often works better with RNNs: Layer Normalization.
- This idea was introduced by Jimmy Lei Ba *et al* (2016): it is very similar to Batch Normalization, but instead of normalizing across the batch dimension, it normalizes across the **features** or **inputs (to a layer)** dimension.
- → **per layer** mean and standard deviation

$$\mathbf{a}^t = W_{hh} h^{t-1} + W_{xh} \mathbf{x}^t$$

$$\mathbf{h}^t = f\left[\frac{\mathbf{g}}{\sigma^t} \odot (\mathbf{a}^t - \mu^t) + \mathbf{b}\right] \qquad \mu^t = \frac{1}{H}\sum_{i=1}^{H} a_i^t \qquad \sigma^t = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^t - \mu^t)^2}$$

element wise multiplication

- Layer Normalization ***learns*** a **scale** and an **offset** parameter for each **hidden state in the layer**.
- In an RNN, it is typically used right **after the linear combination of the inputs and the hidden states, and before the activation function**.
- *Note*: this implementation in Keras for RNN is quite complicated (outside the scope of this module)

Note: If LN after activation function, can use LayerNormalization()

- One advantage is that it can compute the required statistics on the fly, at **each time step, independently for each instance**. This also means that it behaves the same way during training and testing (as opposed to BN), and it does not need to use exponential moving averages to estimate the feature statistics across all instances in the training set.

Long Short Term Memory (LSTM) Hochreiter & Schmidhuber (1997)

- **Motivation:**
- Due to the transformations that the data goes through when traversing an RNN, some information is lost at each time step.
- After a while, the RNN's state contains virtually no trace of the first inputs.
- To tackle this problem, various types of cells with long-term memory have been introduced.
- They have proven so successful that the basic cells are not used much anymore.
- The most popular of these long-term memory cells is the LSTM cell.

- **LSTM:** The key idea is that the network can learn what to store in the long-term state $\mathbf{c}_t$, what to throw away, and what to read from it
  - use forget gate to drop, then add what input gate selected
  - transformed long term state is combined with output to form short term state $\mathbf{h}_t$ (equal to output $\mathbf{y}_t$)

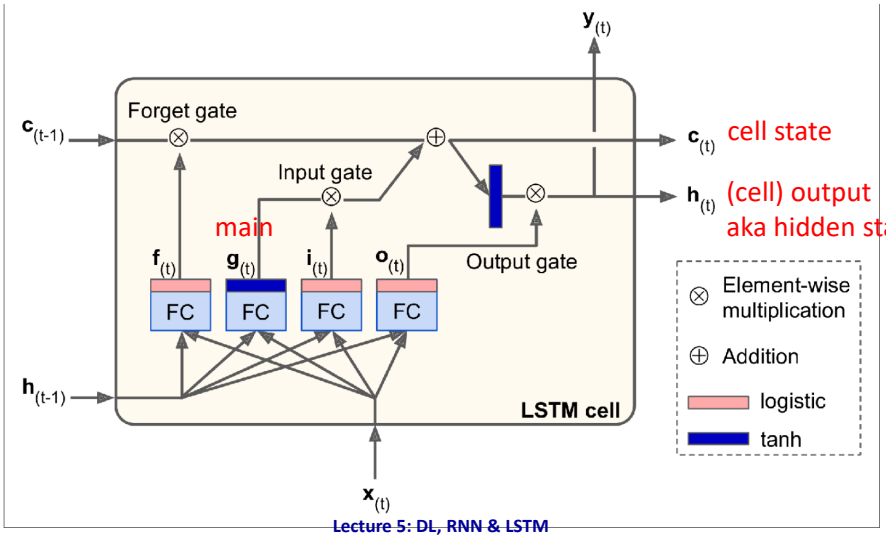
Unstable Gradients solved in LSTM

- We have seen that
  - RNNs suffer from vanishing gradients and caused by long series of multiplications of small values, diminishing the gradients and causing the learning process to become degenerate.
  - In an analogous way, RNNs suffer from exploding gradients affected from large gradient values and hampering the learning process.
- LSTMs solve the problem using a **unique additive gradient structure that includes direct access to the forget gate's activations**, enabling the network to encourage desired behaviour from the error gradient using frequent gates updates on every time step of the learning process.

LSTM

So how does an LSTM cell work? Its architecture is shown in Figure 15-9.

If you don't look at what's inside the box, the LSTM cell looks exactly like a regular cell, except that its state is split into two vectors: $\mathbf{h}_{(t)}$ and $\mathbf{c}_{(t)}$ ("c" stands for "cell"). You can think of $\mathbf{h}_{(t)}$ as the short-term state and $\mathbf{c}_{(t)}$ as the long-term state.



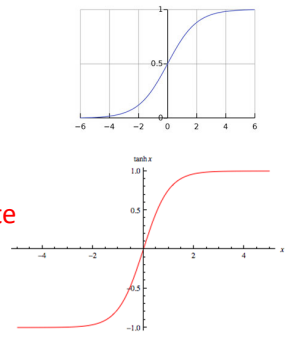


Figure 15-9. LSTM cell

LSTM

Now let's open the box! The key idea is that the network can learn what to store in the long-term state, what to throw away, and what to read from it. As the long-term state $\mathbf{c}_{(t-1)}$ traverses the network from left to right, you can see that it first goes through a *forget gate*, dropping some memories, and then it adds some new memories via the addition operation (which adds the memories that were selected by an *input gate*). The result $\mathbf{c}_{(t)}$ is sent straight out, without any further transformation. So, at each time step, some memories are dropped and some memories are added. Moreover, after the addition operation, the long-term state is copied and passed through the tanh function, and then the result is filtered by the *output gate*. This produces the short-term state $\mathbf{h}_{(t)}$ (which is equal to the cell's output for this time step, $\mathbf{y}_{(t)}$). Now let's look at where new memories come from and how the gates work.

LSTM

element-wise multiplication operations, so if they output 0s they close the gate, and if they output 1s they open it. Specifically:

— The *forget gate* (controlled by $\mathbf{f}_{(t)}$) controls which parts of the long-term state should be erased.

— The *input gate* (controlled by $\mathbf{i}_{(t)}$) controls which parts of $\mathbf{g}_{(t)}$ should be added to the long-term state.

— Finally, the *output gate* (controlled by $\mathbf{o}_{(t)}$) controls which parts of the long-term state should be read and output at this time step, both to $\mathbf{h}_{(t)}$ and to $\mathbf{y}_{(t)}$.

LSTM

Equation 15-3. LSTM computations

$$\mathbf{i}_{(t)} = \sigma\left(\mathbf{W}_{xi}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hi}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_i\right)$$
$$\mathbf{f}_{(t)} = \sigma\left(\mathbf{W}_{xf}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hf}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_f\right)$$
$$\mathbf{o}_{(t)} = \sigma\left(\mathbf{W}_{xo}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{ho}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_o\right)$$
$$\mathbf{g}_{(t)} = \tanh\left(\mathbf{W}_{xg}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hg}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_g\right)$$
$$\mathbf{c}_{(t)} = \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)}$$
$$\mathbf{y}_{(t)} = \mathbf{h}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh\left(\mathbf{c}_{(t)}\right)$$

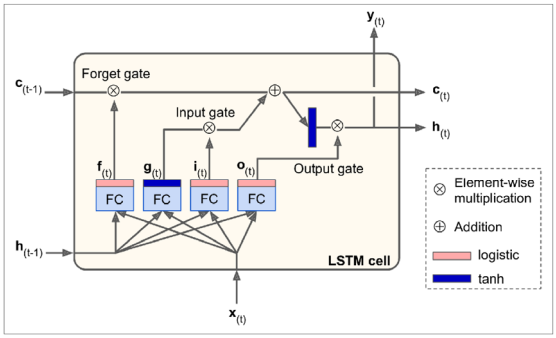


Figure 15-9. LSTM cell

In this equation:

- $\mathbf{W}_{xi}$, $\mathbf{W}_{xf}$, $\mathbf{W}_{xo}$, $\mathbf{W}_{xg}$ are the weight matrices of each of the four layers for their connection to the input vector $\mathbf{x}_{(t)}$.
- $\mathbf{W}_{hi}$, $\mathbf{W}_{hf}$, $\mathbf{W}_{ho}$, and $\mathbf{W}_{hg}$ are the weight matrices of each of the four layers for their connection to the previous short-term state $\mathbf{h}_{(t-1)}$.
- $\mathbf{b}_i$, $\mathbf{b}_f$, $\mathbf{b}_o$, and $\mathbf{b}_g$ are the bias terms for each of the four layers. Note that TensorFlow initializes $\mathbf{b}_f$ to a vector full of 1s instead of 0s. This prevents forgetting everything at the beginning of training.

Gated Recurrent Unit (GRU) cell

Cho *et al* (2014)

simplification of LSTM

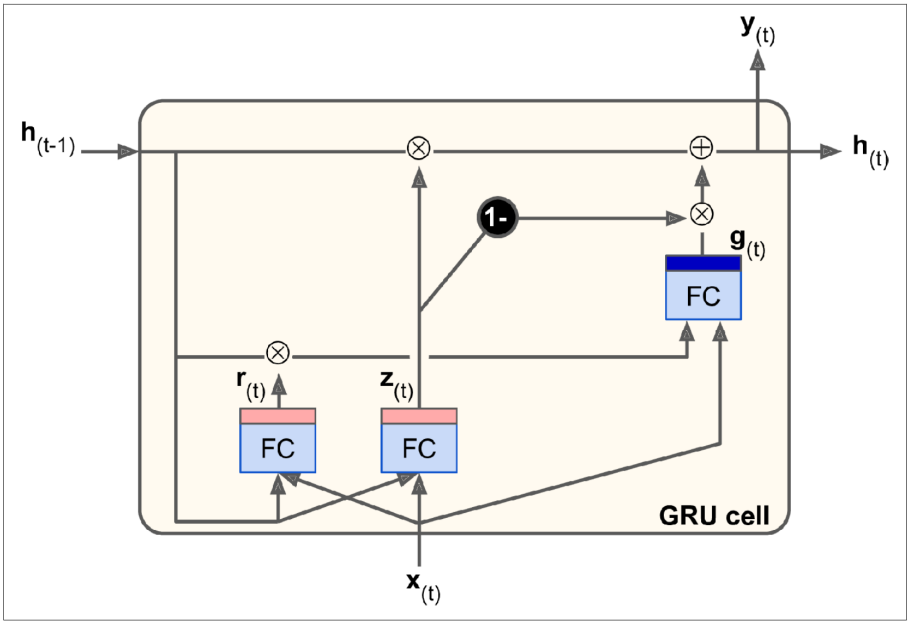


Figure 15-10. GRU cell

GRU

The GRU cell is a simplified version of the LSTM cell, and it seems to perform just as well[12] (which explains its growing popularity). These are the main simplifications:

- Both state vectors are merged into a single vector $\mathbf{h}_{(t)}$.
- A single gate controller $\mathbf{z}_{(t)}$ controls both the forget gate and the input gate. If the gate controller outputs a 1, the forget gate is open (= 1) and the input gate is closed (1 – 1 = 0). If it outputs a 0, the opposite happens. In other words, whenever a memory must be stored, the location where it will be stored is erased first. This is actually a frequent variant to the LSTM cell in and of itself.
- There is no output gate; the full state vector is output at every time step. However, there is a new gate controller $\mathbf{r}_{(t)}$ that controls which part of the previous state will be shown to the main layer ($\mathbf{g}_{(t)}$).

GRU

Equation 15-4. GRU computations

$$\mathbf{z}_{(t)} = \sigma\left(\mathbf{W}_{xz}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hz}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_z\right)$$

$$\mathbf{r}_{(t)} = \sigma\left(\mathbf{W}_{xr}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hr}^{\top}\mathbf{h}_{(t-1)} + \mathbf{b}_r\right)$$

$$\mathbf{g}_{(t)} = \tanh\left(\mathbf{W}_{xg}^{\top}\mathbf{x}_{(t)} + \mathbf{W}_{hg}^{\top}\left(\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}\right) + \mathbf{b}_g\right)$$

$$\mathbf{h}_{(t)} = \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + \left(1 - \mathbf{z}_{(t)}\right) \otimes \mathbf{g}_{(t)}$$

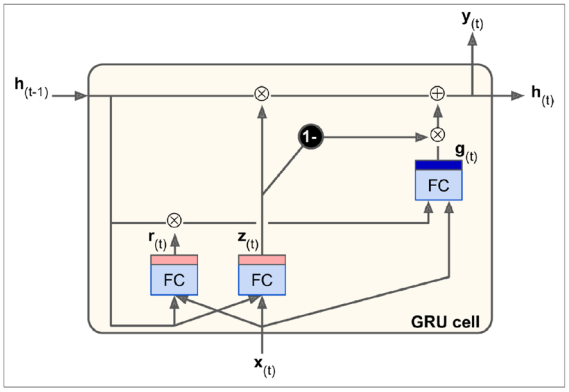


Figure 15-10. GRU cell

Thank You

Questions?

Assoc Prof Tham Chen Khong

E-mail: eletck@nus.edu.sg