جـامعة
المَـنارة
MANARA UNIVERSITY

الغاية من الجلسة :

- التعرف على lstm& GRU
- التعرف على أنواع LAYERS IN KERAS

RNN,GRU & LSTM

(RNN) are designed to work with sequential data. Sequential data(can be time-series) can be in form of text, audio, video etc.

RNN uses the previous information in the sequence to produce the current output. To understand this better I'm taking an example sentence.

"My class is the best class."

At the time(T0), the first step is to feed the word "My" into the network. the RNN produces an output.

At the time(T1), then at the next step we feed the word "class" and the activation value from the previous step. Now the RNN has information of both words "My" and "class".

And this process goes until all words in the sentence are given input. d

What is vanishing gradient problem?

In RNN to train the network you backpropagate through time, at each step the gradient is calculated. The gradient is used to update weights in the network. If the effect of the previous layer on the current layer is small then the gradient value will be small and vice-versa. If the gradient of the previous layer is smaller then the gradient of the current layer will be even smaller. This makes the gradients exponentially shrink down as we backpropagate. Smaller gradient means it will not affect the weight updation. Due to this, the network does not learn the effect of earlier inputs. Thus, causing the short-term memory problem.

Solution for vanishing gradient

To overcome this problem two specialised versions of RNN were created. They are GRU(Gated Recurrent Unit) LSTM(Long Short Term Memory). Suppose there are 2 sentences. Sentence one is "My cat is ….. she was ill.", the second one is "The cats ….. they were ill." At the ending of the sentence, if we need to predict the word "was" / "were" the network has to remember the starting word "cat"/"cats". So, LSTM's and GRU's make use of memory cell to store the activation value of previous words in the long sequences. Now the concept of gates come into the picture. Gates are used for controlling the flow of information in the network. Gates are capable of learning which inputs in the sequence are important and store their information in the memory unit. They can pass the information in long sequences and use them to make predictions.

Gated Recurrent Units The workflow of GRU is same as RNN but the difference is in the operations inside the GRU unit.

Gates are nothing but neural networks, each gate has its own weights and biases(but don't forget that weights and bias for all nodes in one layer are same). **Update gate**

Update gate decides if the cell state should be updated with the candidate state(current activation value)or not.

Relevence gate

The reset gate is used to decide whether the previous cell state is important or not. Sometimes the reset gate is not used in simple GRU.

In GRU the final cell state is directly passing as the activation to the next cell.

In GRU,

If reset close to 0, ignore previous hidden state (allows the model to drop information that is irrelevant in the future).

If gamma(update gate) close to 1, then we can copy information in that unit through many steps!

Gamma Controls how much of past state should matter now.

Long Short-Term Memory

Now you know about RNN and GRU, so let's quickly understand how LSTM works in brief. LSTMs are pretty much similar to GRU's, they are also intended to solve the vanishing gradient problem. Additional to GRU here there are 2 more gates forget gate, output gate. From GRU, you already know about all other operations except forget gate and output gate.

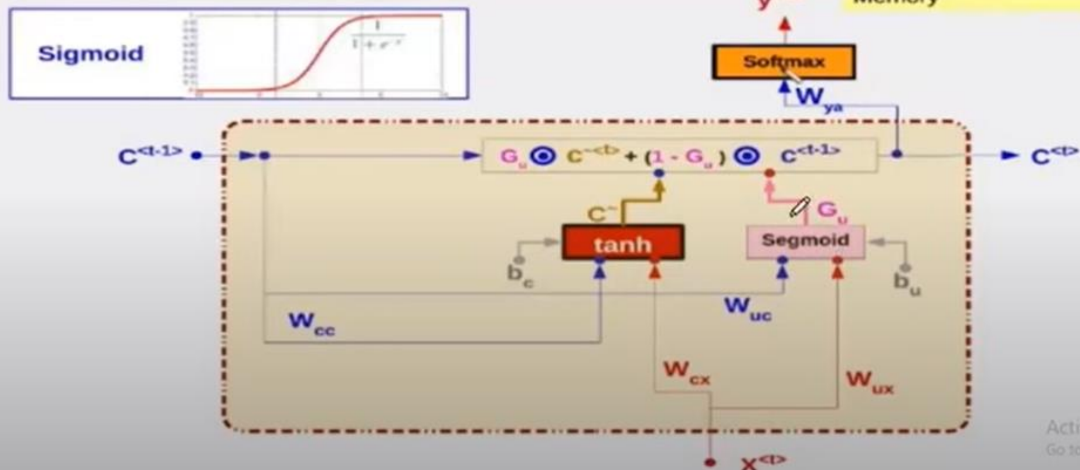
All 3 gates(input gate, output gate, forget gate) use sigmoid as activation function so all gate values are between 0 and 1.

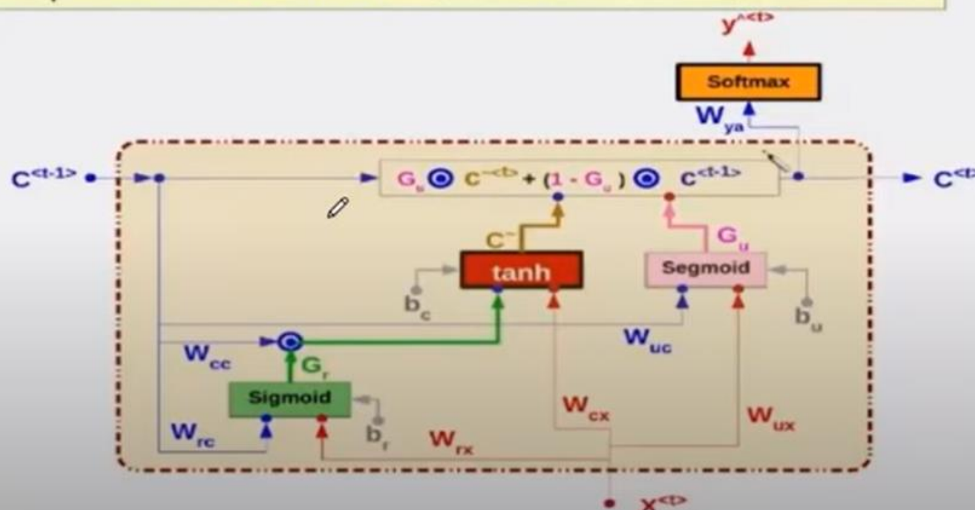
Forget gate

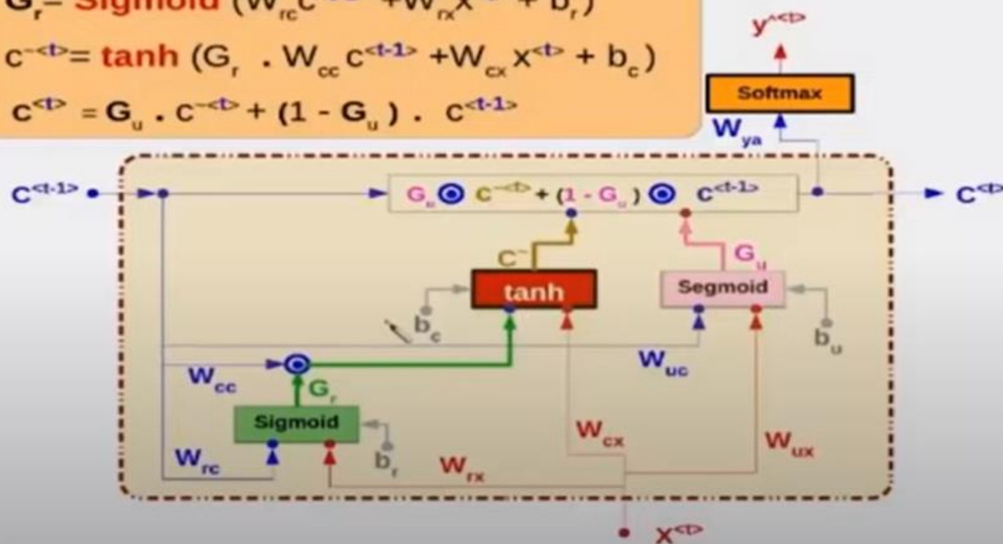
It controls what is kept vs forgotten, from previous cell state. In laymen terms, it will decide how much information from the previous state should be kept and forget remaining.

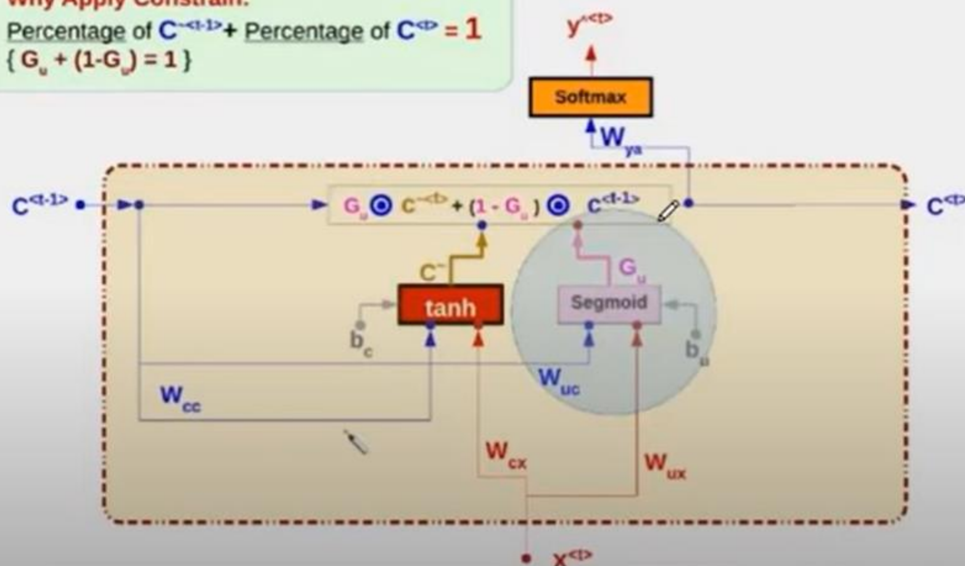
Output gate

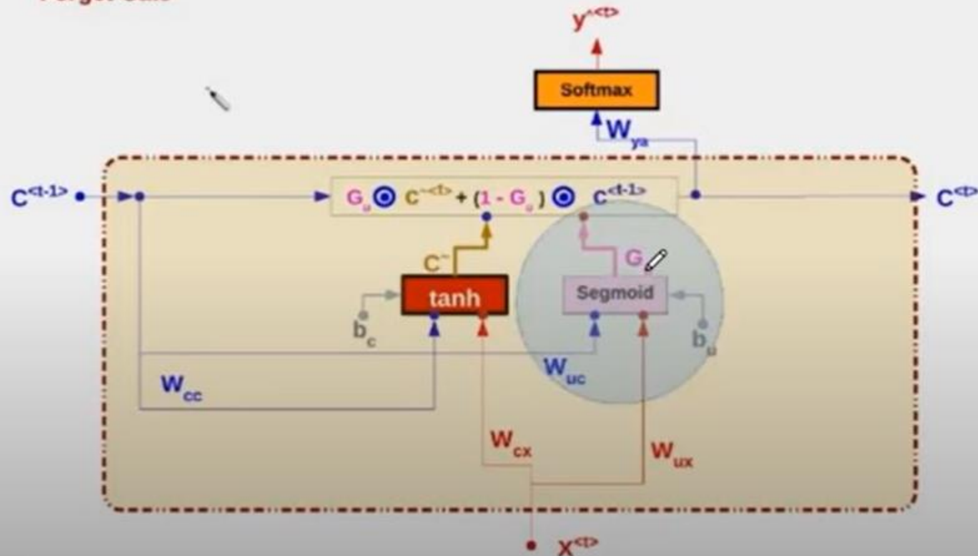
It controls which parts of the cell are output to the hidden state. It will determine what the next hidden state will be.

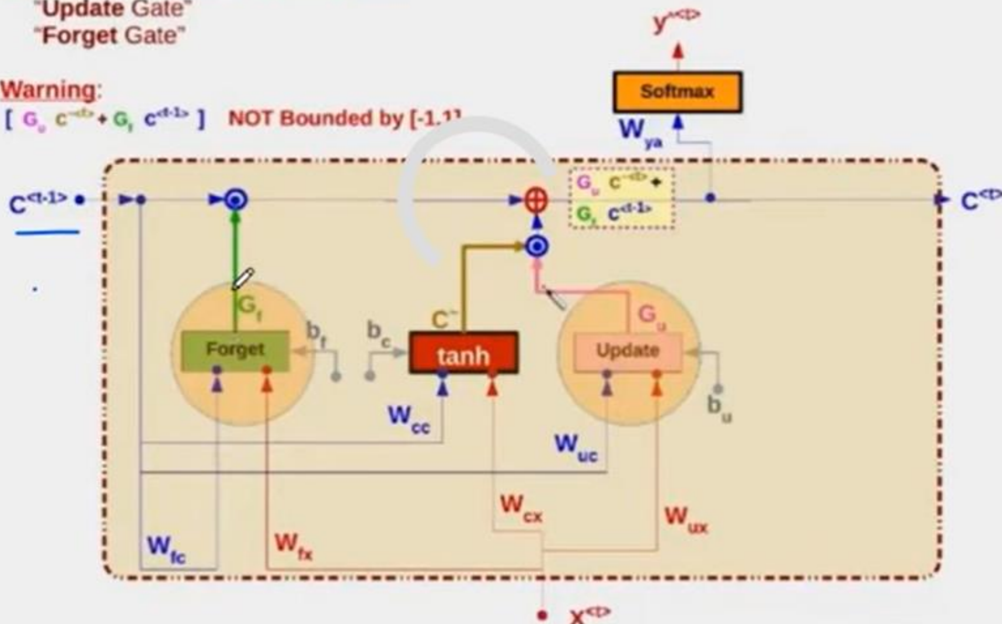

From RNN to GRU
[1] Adding Update Gate Gu
If Gu = 0 , Keep Memory Value "C<t>" Same as Previous Value "C<t-1>"
If Gu = 1 , Forget Previous Memory Value "C<t-1>"
C~ Candidate Value of Updated Memory
C Value of Updated Memory
Sigmoid
y^<t>
Softmax
Wya
C<t-1>
Gu ⊙ C~<t> + (1 - Gu) ⊙ C<t-1>
C<t>
C~
tanh
Gu
Segmoid
bc
bu
Wcc
Wuc
Wcx
Wux
x<t>


From RNN to GRU
[2] Adding Relevance Gate Gr
If Gr = 1 , C<t-1> is Relevant to update Candidate Memory cell value "C~"
If Gr = 0 , C<t-1> is IrRelevant to update Candidate Memory cell value "C~"
y^<t>
Softmax
Wya
C<t-1>
Gu ⊙ C~<t> + (1 - Gu) ⊙ C<t-1>
C<t>
C~
tanh
Gu
Segmoid
bc
bu
Wcc
Gr
Sigmoid
Wuc
Wcx
Wux
Wrc
br
Wrx
x<t>

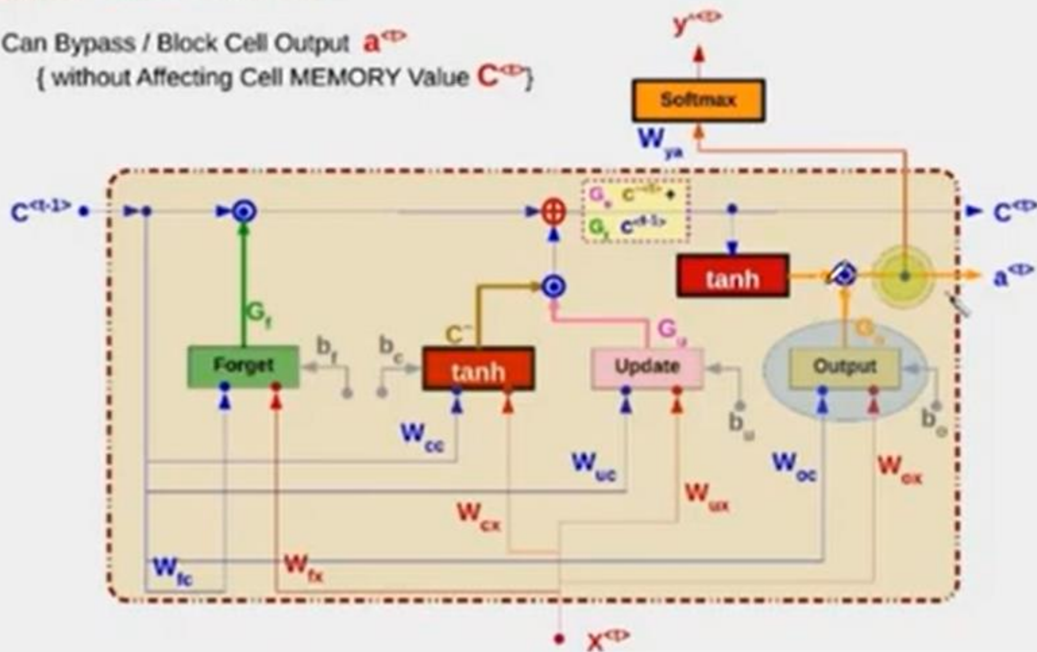

From RNN to GRU
G_u= Sigmoid $(W_{uc}c^{<t-1>} + W_{ux}x^{<t>} + b_u)$
G_r= Sigmoid $(W_{rc}c^{<t-1>} + W_{rx}x^{<t>} + b_r)$
$\tilde{c}^{<t>}$= tanh $(G_r \cdot W_{cc}c^{<t-1>} + W_{cx}x^{<t>} + b_c)$
$c^{<t>} = G_u \cdot \tilde{c}^{<t>} + (1 - G_u) \cdot c^{<t-1>}$
$y^{<t>}$
Softmax
W_{ya}
$c^{<t-1>}$
$G_u \odot \tilde{c}^{<t>} + (1 - G_u) \odot c^{<t-1>}$
$c^{<t>}$
$\tilde{C}$
tanh
G_u
Segmoid
b_c
b_u
W_{uc}
W_{cc}
G_r
Sigmoid
W_{rc}
b_r
W_{rx}
W_{cx}
W_{ux}
$x^{<t>}$

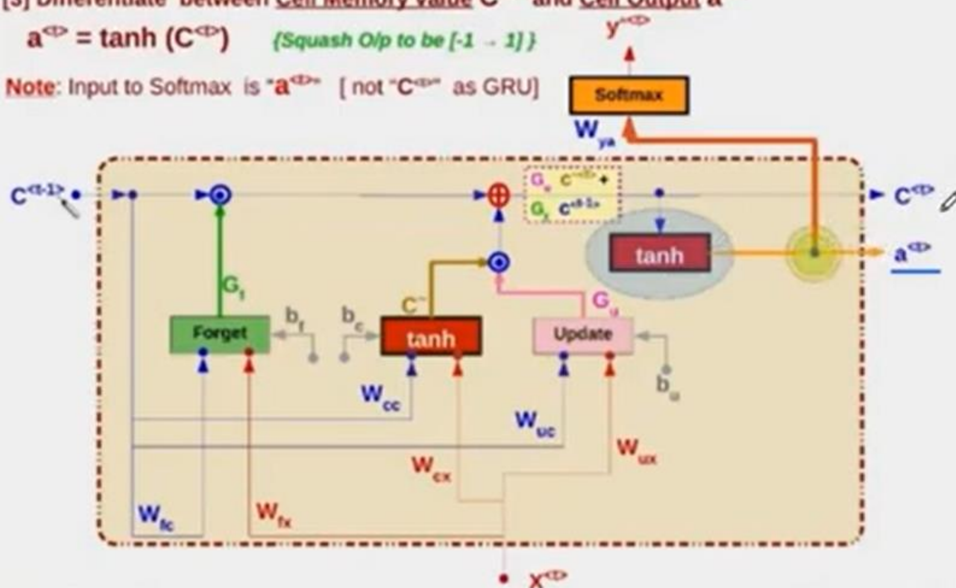

From GRU to LSTM
[2] ??????
Why Apply Constrain:
Percentage of $C^{<t-1>}$+ Percentage of $C^{<t>}$ = 1
$\{ G_u + (1-G_u) = 1 \}$
$y^{<t>}$
Softmax
W_{ya}
$c^{<t-1>}$
$G_u \odot \tilde{c}^{<t>} + (1 - G_u) \odot c^{<t-1>}$
$c^{<t>}$
$\tilde{C}$
tanh
G_u
Segmoid
b_c
b_u
W_{uc}
W_{cc}
W_{cx}
W_{ux}
$x^{<t>}$


From GRU to LSTM
[2] Split "Update Gate" into two gates:
"Update Gate"
"Forget Gate"
Softmax
tanh
Segmoid


From GRU to LSTM
[2] Split "Update Gate" into two gates:
"Update Gate"
"Forget Gate"
Warning:
NOT Bounded by [-1,1]
Softmax
Forget
tanh
Update

جَـامعة
المَـنارة
MANARA UNIVERSITY





Keras - Layers

A Keras layer requires shape of the input (input_shape) to understand the structure of the input data, initializer to set the weight for each input and finally activators to transform the output to make it non-linear. In between, constraints restricts and specify the range in which the weight of input data to be generated and regularizer will try to optimize the layer (and the model) by dynamically applying the penalties on the weights during optimization process.

To summarise, Keras layer requires below minimum details to create a complete layer.

---

axis represent the dimension in which the constraint to be applied. e.g. in Shape (2,3,4) axis 0 denotes first dimension, 1 denotes second dimension and 2 denotes third dimension

MinMaxNorm

Constrains weights to be norm between specified minimum and maximum values.

```
my_constrain = constraints.MinMaxNorm(min_value = 0.0, max_value = 1.0, rate = 1.0, axis = 0)
```

where, **rate** represent the rate at which the weight constrain is applied.

Keras - Convolution Layers

Keras contains a lot of layers for creating Convolution based ANN, popularly called as Convolution Neural Network (CNN). All convolution layer will have certain properties (as listed below), which differentiate it from other layers (say Dense layer).

Filters − It refers the number of filters to be applied in the convolution. It affects the dimension of the output shape.

kernel size − It refers the length of the convolution window.

Strides − It refers the stride length of the convolution.

Padding − It refers the how padding needs to be done on the output of the convolution. It has three values which are as follows −

valid means no padding

causal means causal convolution.

same means the output should have same length as input and so, padding should be applied accordingly

```
keras.layers.Conv1D(
   filters,
   kernel_size,
   strides = 1,
   padding = 'valid',
   data_format = 'channels_last',
   dilation_rate = 1,
   activation = None,
   use_bias = True,
   kernel_initializer = 'random_uniform',
   bias_initializer = 'zeros',
   kernel_regularizer = None,
   bias_regularizer = None,
   activity_regularizer = None,
   kernel_constraint = None,
   bias_constraint = None
)

keras.layers.Conv2D
   (filters, kernel_size,
   strides = (1, 1),
   padding = 'valid',
   data_format = None,
   dilation_rate = (1, 1),
   activation = None,
   use_bias = True,
   kernel_initializer = 'glorot_uniform',
   bias_initializer = 'zeros',
   kernel_regularizer = None,
   bias_regularizer = None,
   activity_regularizer = None,
   kernel_constraint = None,
   bias_constraint = None
)
```

Flatten Layer

Flatten Layer is used to flatten the input. For example, if flatten is applied to layer having input shape as (batch_size, 2,2), then the output shape of the layer will be (batch_size, 4)

جامعة المنارة
MANARA UNIVERSITY

```
from keras.models import Sequential
from keras.layers import Activation, Dense, Flatten


 model = Sequential()
 layer_1 = Dense(16, input_shape=(8,8))
 model.add(layer_1)
 layer_2 = Flatten()
 model.add(layer_2)
 layer_2.input_shape (None, 8, 16)
```

where, the second layer input shape is (None, 8, 16) and it gets flattened into (None, 128)

Keras - Reshape Layers Reshape is used to change the shape of the input. For example, if reshape with argument (2,3) is applied to layer having input shape as (batch_size, 3, 2), then the output shape of the layer will be (batch_size, 2, 3)

```
model = Sequential()
layer_1 = Dense(16, input_shape = (8,8))
model.add(layer_1)
layer_2 = Reshape((16, 8))
model.add(layer_2)
layer_2.input_shape (None, 8, 16)
layer_2.output_shape (None, 16, 8)
```

Pooling layers

It is used to perform max pooling operations on temporal data. The signature of the MaxPooling1D function and its arguments with default value is as follows

```
keras.layers.MaxPooling1D (
   pool_size = 2,
   strides = None,
   padding = 'valid',
   data_format = 'channels_last'
)
```

Embedding layer

It performs embedding operations in input layer. It is used to convert positive into dense vectors of fixed size. Its main application is in text