RNN, GRU and LSTM

Note :
To read about mathematical calculation used in LTSM

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

and GRU

https://en.wikipedia.org/wiki/Hadamard_product_(matrices)

Recurrent Neural Networks

Recurrent Neural Networks (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 consider following 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. You can see the animation below to visualize and understand.

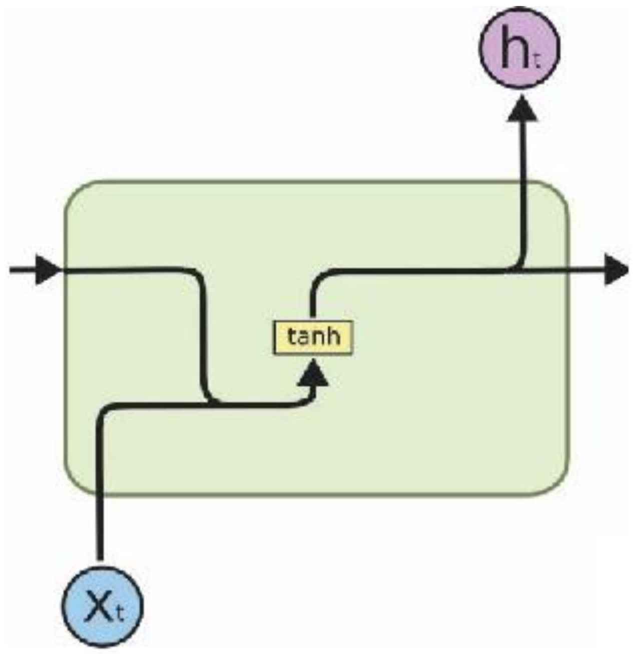
The workflow of RNN.

At the **last step**, the RNN has information about all the previous words.

Note: In RNN *weights and bias for all the nodes in the layer are same.*

Architecture of the RNN unit.

It takes input from the previous step and current input. Here *tanh* is the activation function, instead of *tanh* you can use other activation function as well.



RNN basic architecture

💡RNN's face short-term memory problem. It is caused due to vanishing gradient problem. As RNN processes more steps it suffers from vanishing gradient more than other neural network architectures.

Q. What is vanishing gradient problem?

Ans: 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.

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

The hidden state of RNN

The main problem is that ***it's too difficult for RNN to learn to preserve information over many timesteps****. In vanilla RNN the hidden state is constantly being* ***rewritten.***

How about an RNN with a separate memory?

Solution for vanishing gradient

To overcome this problem two specialised versions of RNN were created.

1) GRU(Gated Recurrent Unit)
   Mathematics :
   https://en.wikipedia.org/wiki/Hadamard_product_(matrices)
2) LSTM(Long Short Term Memory).
   Mathematics
   http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Suppose there are 2 sentences. Sentence one is

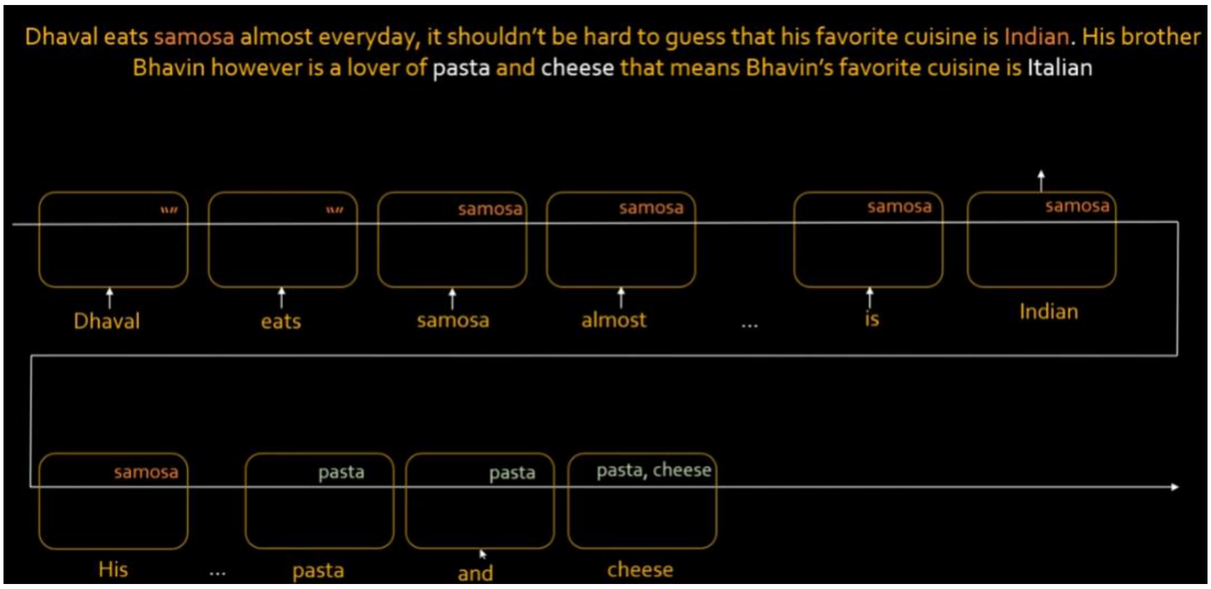
"My ***cat*** is ...... she ***was*** ill.",

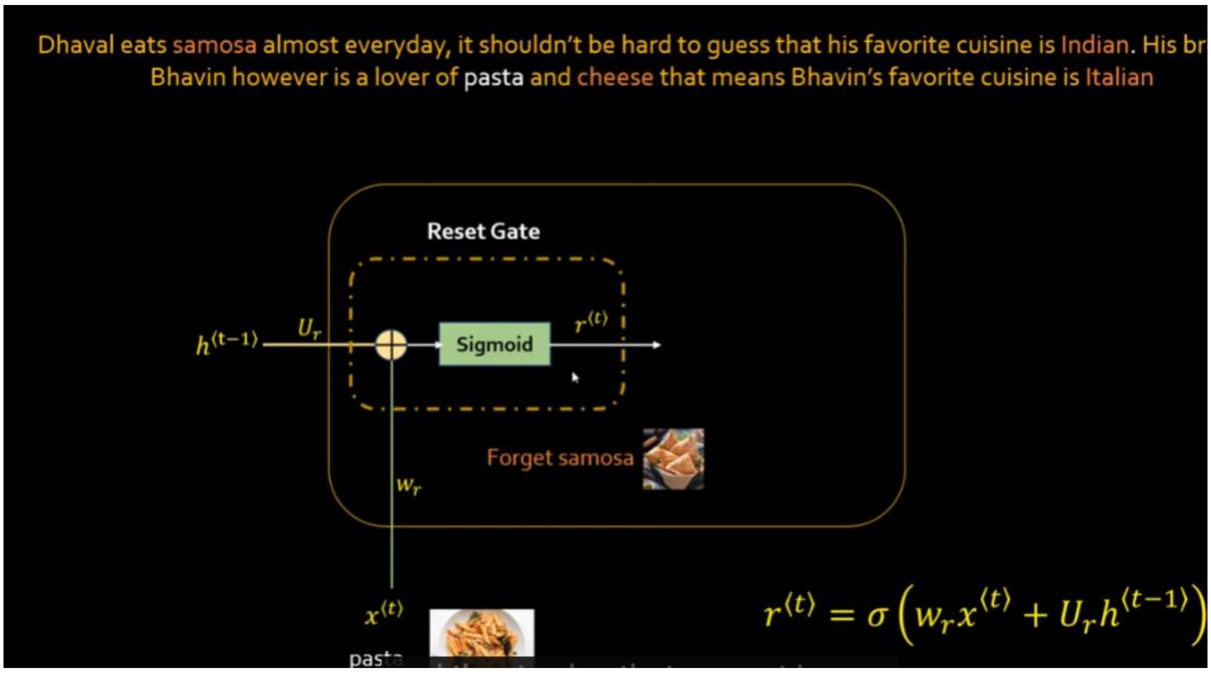
the second one is "The ***cats*** ..... they ***were*** ill."

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. Let's see the architecture of it.

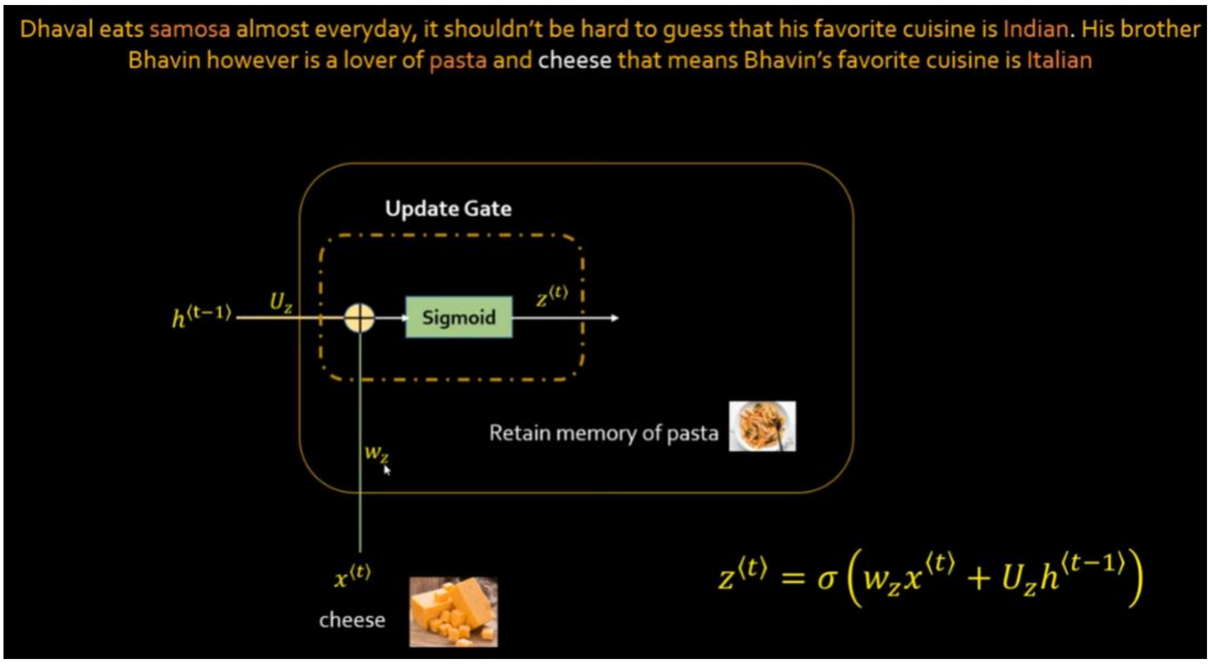

Dhaval eats samosa almost everyday, it shouldn't be hard to guess that his favorite cuisine is Indian. His brother Bhavin however is a lover of pasta and cheese that means Bhavin's favorite cuisine is Italian
samosa
samosa
samosa
samosa
Dhaval
eats
samosa
almost
...
is
Indian
samosa
pasta
pasta
pasta, cheese
His
...
pasta
and
cheese



Dhaval eats samosa almost everyday, it shouldn't be hard to guess that his favorite cuisine is Indian. His br
Bhavin however is a lover of pasta and cheese that means Bhavin's favorite cuisine is Italian

Reset Gate

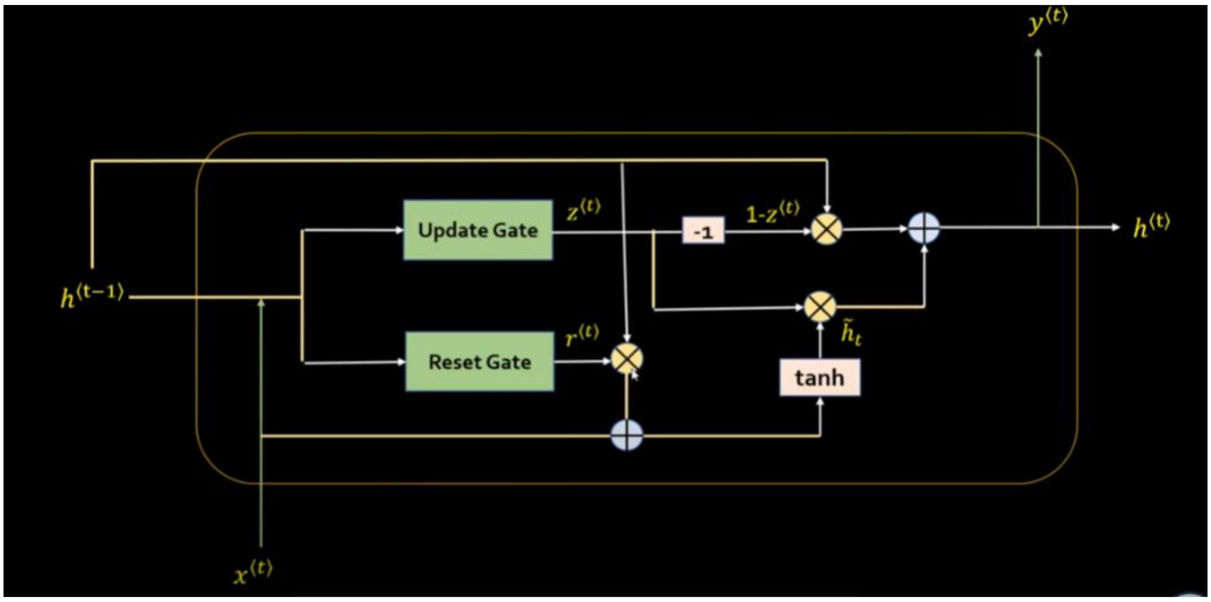
$$r^{\langle t \rangle} = \sigma\left(w_r x^{\langle t \rangle} + U_r h^{\langle t-1 \rangle}\right)$$

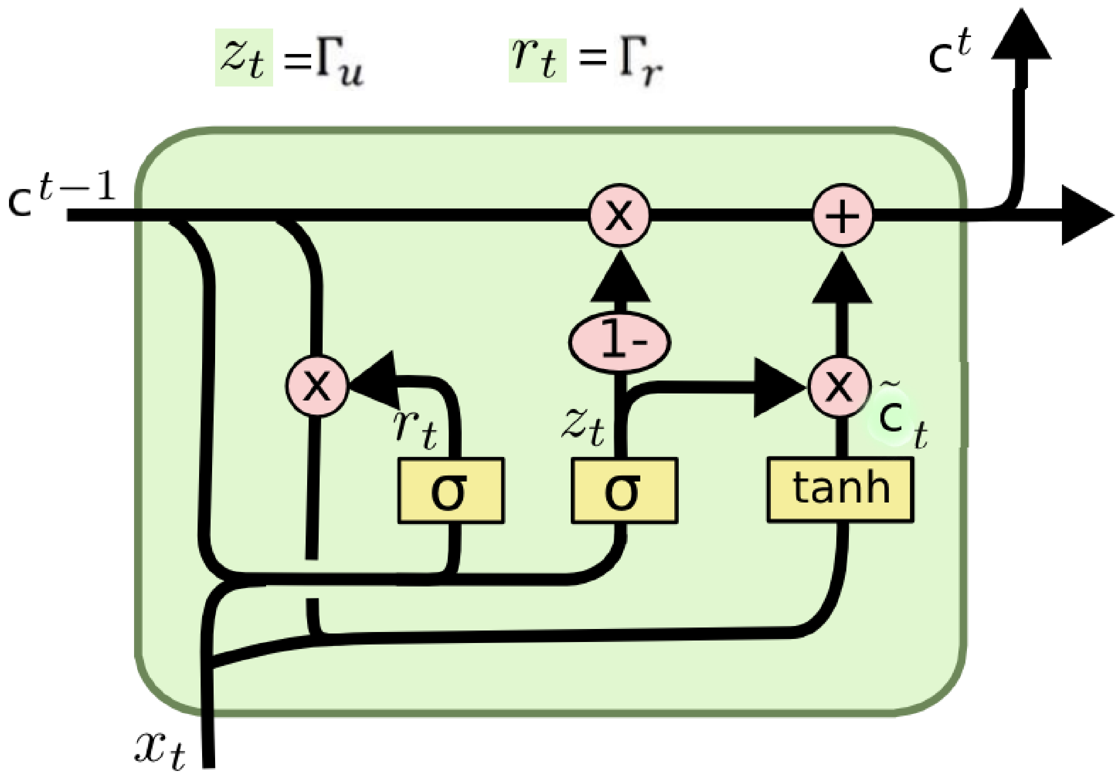


Dhaval eats samosa almost everyday, it shouldn't be hard to guess that his favorite cuisine is Indian. His brother
Bhavin however is a lover of pasta and cheese that means Bhavin's favorite cuisine is Italian

Update Gate

$$z^{\langle t \rangle} = \sigma\left(w_z x^{\langle t \rangle} + U_z h^{\langle t-1 \rangle}\right)$$

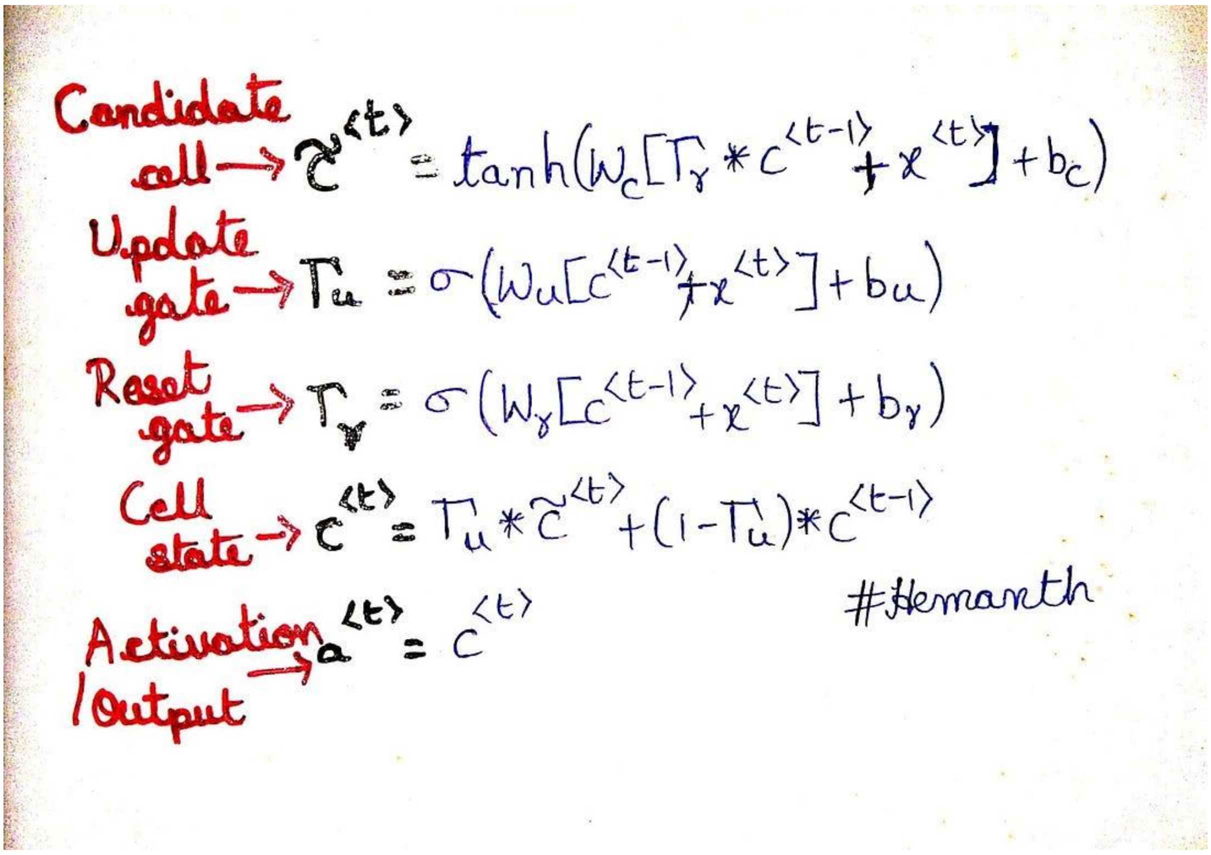




GRU basic architecture

Inside GRU it has two gates 1)reset gate 2)update gate

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).



Candidate cell $\rightarrow \tilde{c}^{<t>} = \tanh(W_c[\Gamma_r * c^{<t-1>} + x^{<t>}] + b_c)$

Update gate $\rightarrow \Gamma_u = \sigma(W_u[c^{<t-1>} + x^{<t>}] + b_u)$

Reset gate $\rightarrow \Gamma_r = \sigma(W_r[c^{<t-1>} + x^{<t>}] + b_r)$

Cell state $\rightarrow c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1-\Gamma_u) * c^{<t-1>}$

Activation /Output $\rightarrow a^{<t>} = c^{<t>}$

#Hemanth

Formulae for gates and cell state of GRU

Update gate

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

Reset 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.

Candidate cell

It is just simply the same as the hidden state(activation) of RNN.

Final cell state

The final cell state is dependent on the update gate. It may or may not be updated with candidate state. Remove some content from last cell state, and write some new cell content.

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

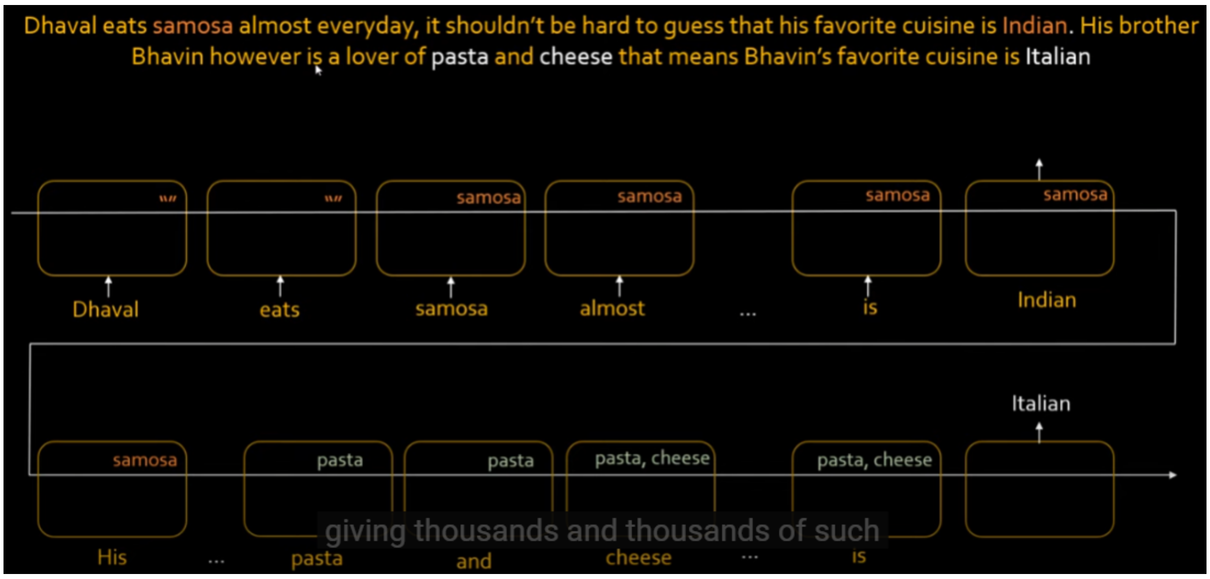
In GRU,

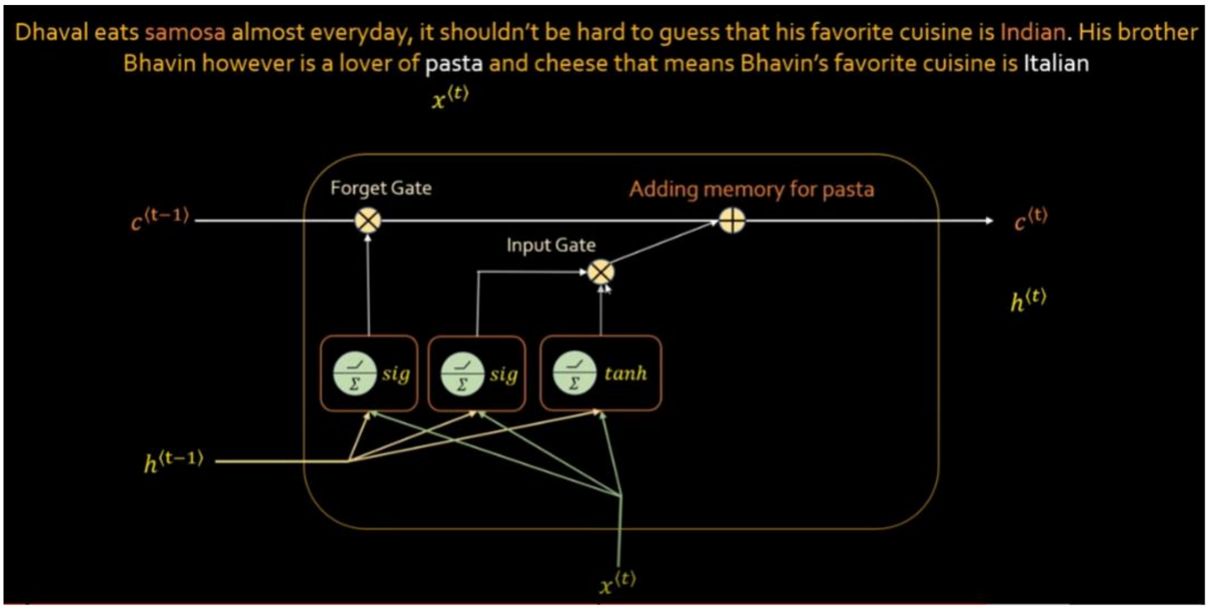
- If reset close to 0, ignore previous hidden state (allows the model to drop information that is irrelevant in the future).
- If gamma(update gate) close to 1, then we can copy information in that unit through many steps!
- Gamma Controls how much of past state should matter now.

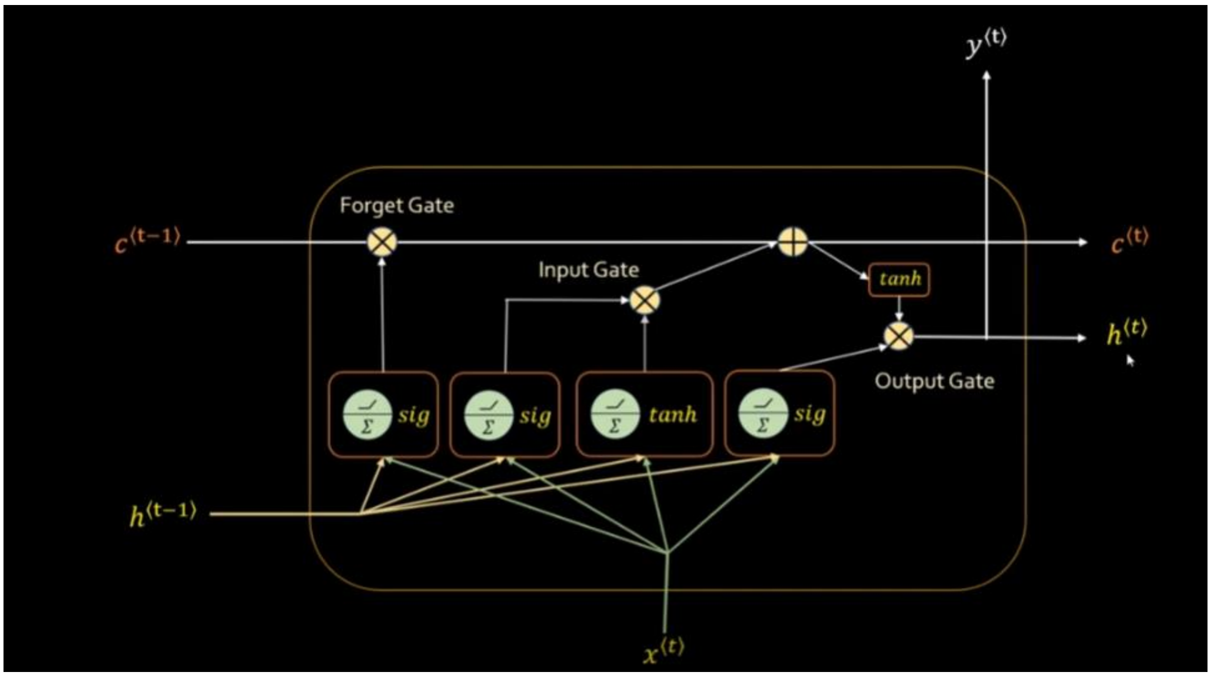
Long Short-Term Memory

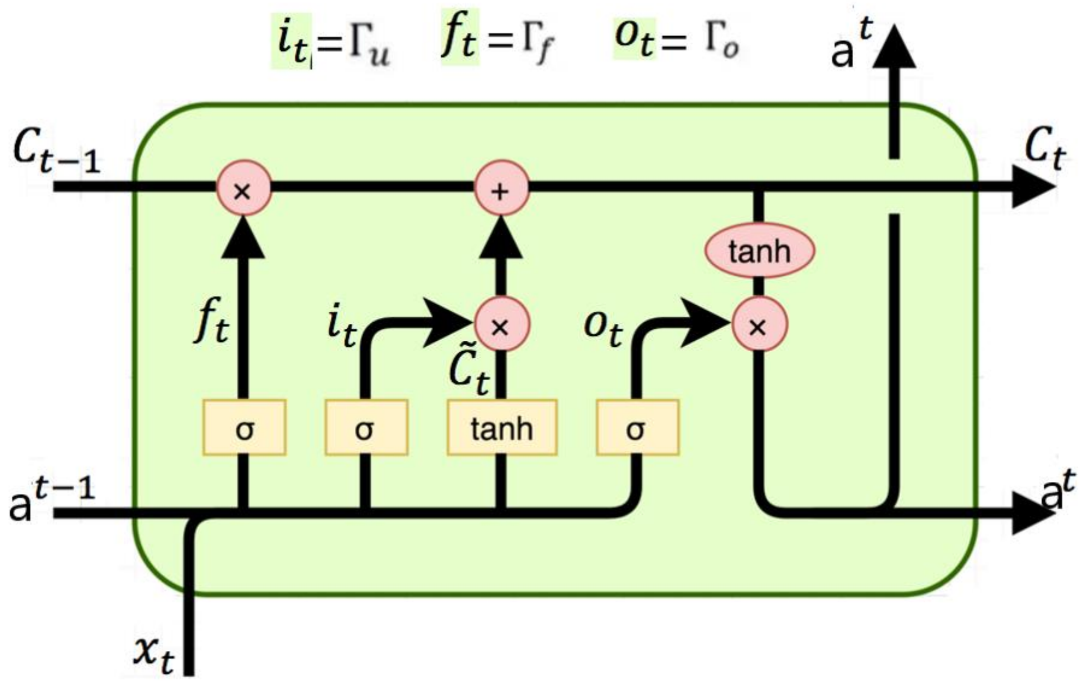
Now you know about RNN and GRU, so let's quickly understand how LSTM works in brief. LSTMs are pretty much similar to GRU's, they are also intended to solve the vanishing gradient problem. Additional to GRU here there are 2 more gates 1)forget gate 2)output gate.

First, look at the architecture of it.


Dhaval eats samosa almost everyday, it shouldn't be hard to guess that his favorite cuisine is Indian. His brother Bhavin however is a lover of pasta and cheese that means Bhavin's favorite cuisine is Italian

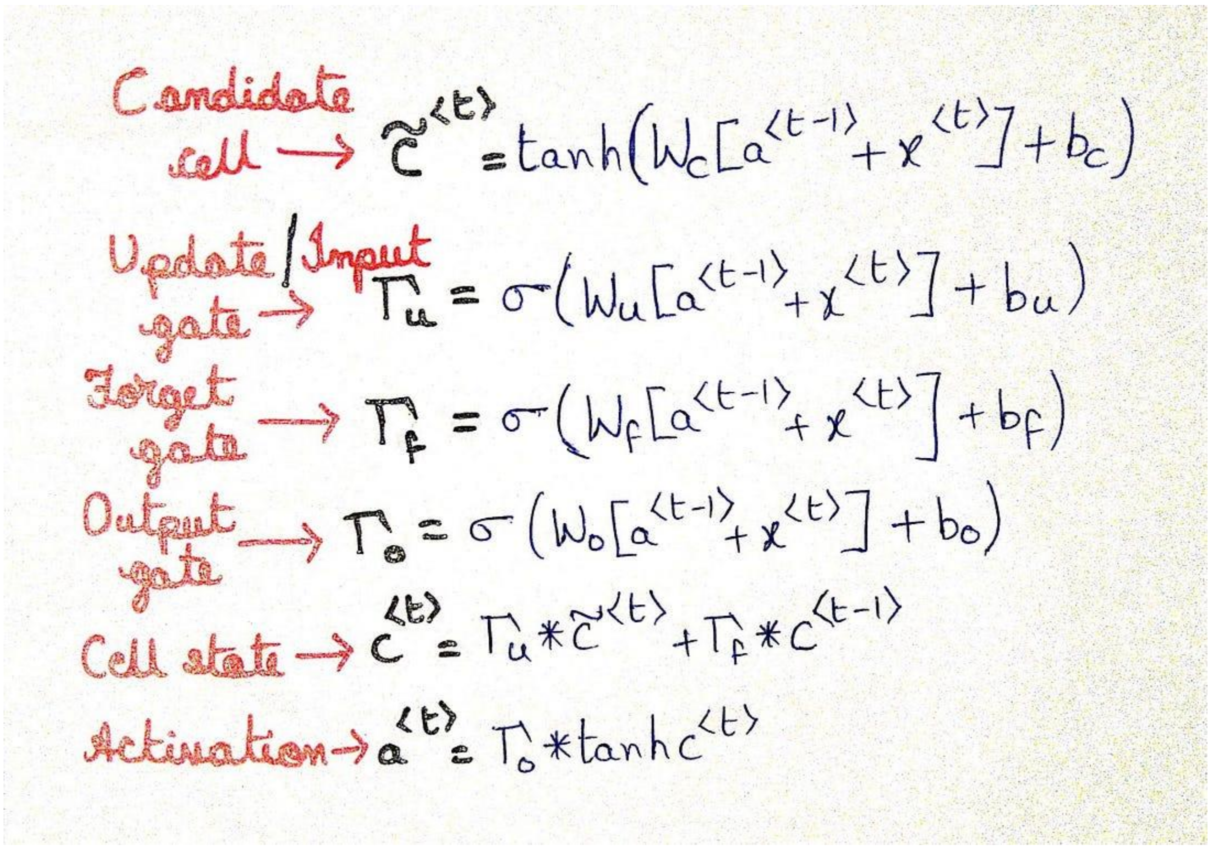

Dhaval eats samosa almost everyday, it shouldn't be hard to guess that his favorite cuisine is Indian. His brother Bhavin however is a lover of pasta and cheese that means Bhavin's favorite cuisine is Italian





LSTM basic architecture

Now, look at the operations inside it.



Candidate cell $\rightarrow \tilde{c}^{\langle t\rangle} = \tanh(W_c[a^{\langle t-1\rangle} + x^{\langle t\rangle}] + b_c)$

Update/Input gate $\rightarrow \Gamma_u = \sigma(W_u[a^{\langle t-1\rangle} + x^{\langle t\rangle}] + b_u)$

Forget gate $\rightarrow \Gamma_f = \sigma(W_f[a^{\langle t-1\rangle} + x^{\langle t\rangle}] + b_f)$

Output gate $\rightarrow \Gamma_o = \sigma(W_o[a^{\langle t-1\rangle} + x^{\langle t\rangle}] + b_o)$

Cell state $\rightarrow c^{\langle t\rangle} = \Gamma_u * \tilde{c}^{\langle t\rangle} + \Gamma_f * c^{\langle t-1\rangle}$

Activation $\rightarrow a^{\langle t\rangle} = \Gamma_o * \tanh c^{\langle t\rangle}$

Formulae for gates and cell state of LSTM

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.

Phew! That's enough theory, now let us start coding.

I'm taking airline passengers dataset and provide the performance of all 3 (RNN, GRU, LSTM) models on the dataset.

My motive is to make you understand and know how to implement these models on any dataset. To make it simple, I'm not focusing on the number of neurons in the hidden layer or number of layers in the network(You can play with these to get better accuracy).

About dataset:

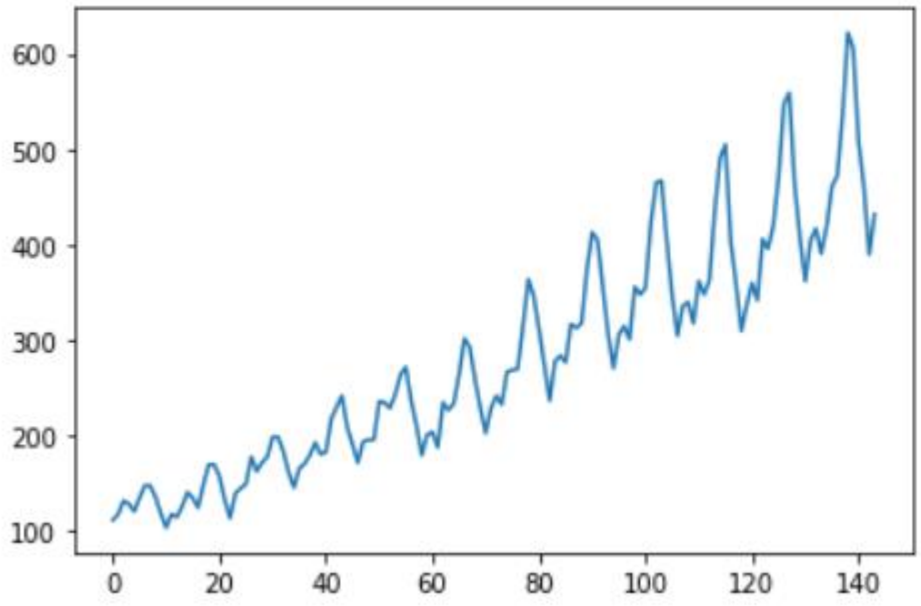
The dataset provides a record of the number of people travelling in US airlines in a particular month. It has a record of 142 months. It has 2 columns "Month" and "No. of Passengers". But in this case, I want to use univariate dataset. Only "No. of Passengers" is used.

Importing all the necessary libraries and dataset.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM,GRU,SimpleRNN
from sklearn.preprocessing import MinMaxScalerdf =
pd.read_csv('airline-passengers.csv')
df.head()
df.drop(['Month'],axis=1,inplace=True)
dataset = np.array(df)
dataset.reshape(-1,1)
```

To visualize the dataset, `plt.plot(dataset)`

```
[<matplotlib.lines.Line2D at 0x22db6945048>]
```



No. of passengers (Y-axis) vs month(X-axis)

It shows that the number of passengers is linearly increasing over the months.

Machine learning model/ Neural network works better if all the data is scaled.

```
scaler = MinMaxScaler()
dataset = scaler.fit_transform(dataset)
```

Divide the data for training and testing. I split the dataset into (75% training and 25% testing). In the dataset, we can estimate the **'i'**th value based on the **'i-1'**th value. You can also increase the length of the input sequence by taking i-1,i-2,i-3… to predict **'i'**th value.

```
train_size = int(len(dataset) * 0.75)
test_size = len(dataset) - train_size
train=dataset[:train_size,:]
test=dataset[train_size:142,:]
def getdata(data,lookback):
    X,Y=[],[]
    for i in range(len(data)-lookback-1):
        X.append(data[i:i+lookback,0])
        Y.append(data[i+lookback,0])
    return np.array(X),np.array(Y).reshape(-1,1)
lookback=1
X_train,y_train=getdata(train,lookback)
X_test,y_test=getdata(test,lookback)
X_train=X_train.reshape(X_train.shape[0],X_train.shape[1],1)
X_test=X_test.reshape(X_test.shape[0],X_test.shape[1],1)
```

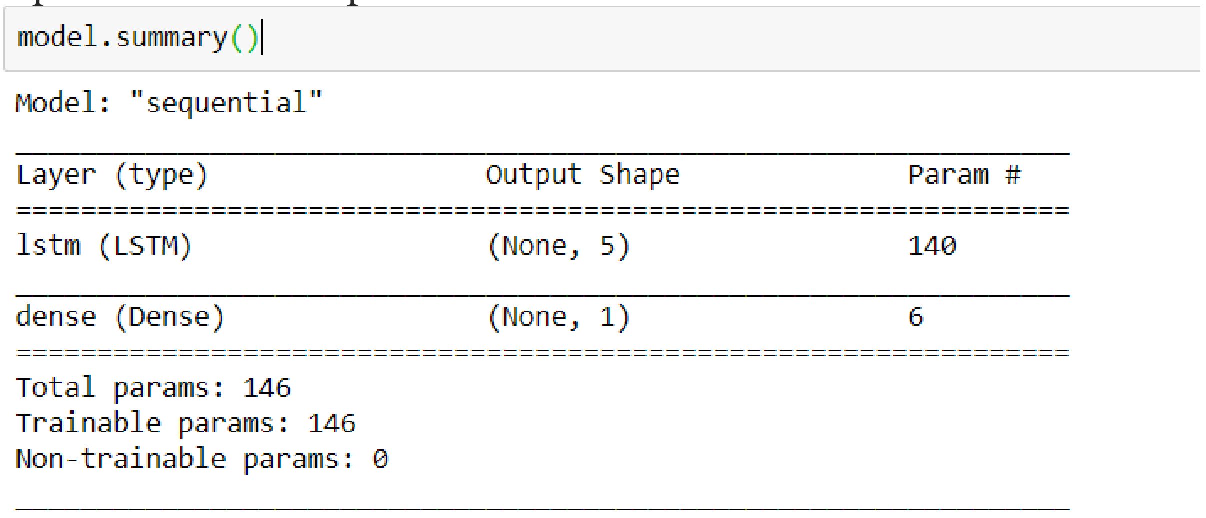
I made the sequential model with only 2 layers. Layers:

1. Simple RNN/GRU/LSTM
2. Dense layer

In this code, I'm using LSTM. You can also use the other two just by replacing "LSTM" with "SimpleRNN"/"GRU" in the below code(line 2).

```
model=Sequential()
model.add(LSTM(5,input_shape=(1,lookback)))
model.add(Dense(1))
model.compile(loss='mean_squared_error',optimizer='adam')
```

In the LSTM layer, I used 5 neurons and it is the first layer (hidden layer) of the neural network, so the input_shape is the shape of the input which we will pass.



```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
lstm (LSTM)                  (None, 5)                 140       
_________________________________________________________________
dense (Dense)                (None, 1)                 6         
=================================================================
Total params: 146
Trainable params: 146
Non-trainable params: 0
_________________________________________________________________
```

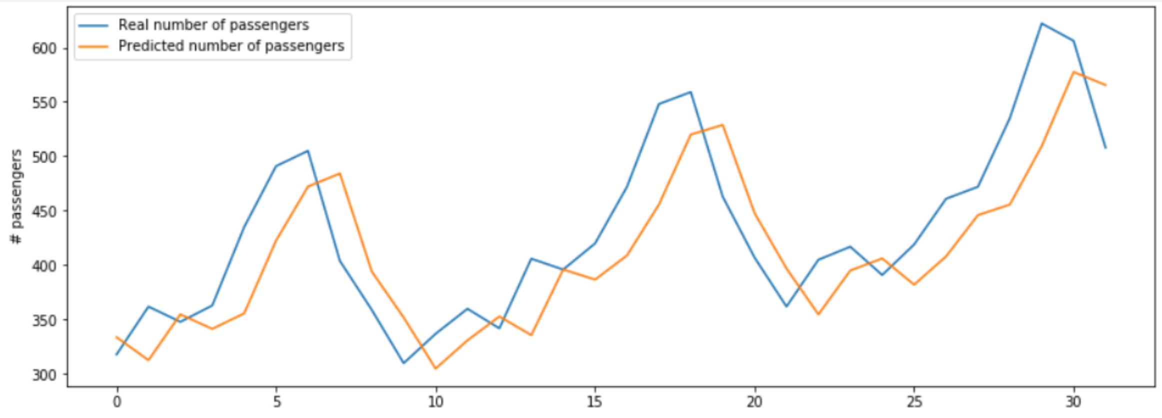
Summary of the neural network

Now, the model is ready. So start training the model.

```
model.fit(X_train, y_train, epochs=50, batch_size=1)
y_pred=model.predict(X_test)
y_test=scaler.inverse_transform(y_test)
y_pred=scaler.inverse_transform(y_pred)
```

Finally, to visualize the real values and predicted result.

```
plt.figure(figsize=(14,5))
plt.plot(y_test, label = 'Real number of passengers')
plt.plot(y_pred, label = 'Predicted number of passengers')
plt.ylabel('# passengers')
plt.legend()
plt.show()
```



Passenger(Y-axis) vs No. of samples(X-axis)

For this dataset and with the simple network by using 50 epochs I got the following mean_squared_error values.

```
from sklearn.metrics import mean_squared_error
mean_squared_error(y_test,y_pred)
```

Simple RNN: 3000

GRU: 2584

LSTM: 2657

To read about mathematical calculation used in LTSM

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

and GRU

https://en.wikipedia.org/wiki/Hadamard_product_(matrices)