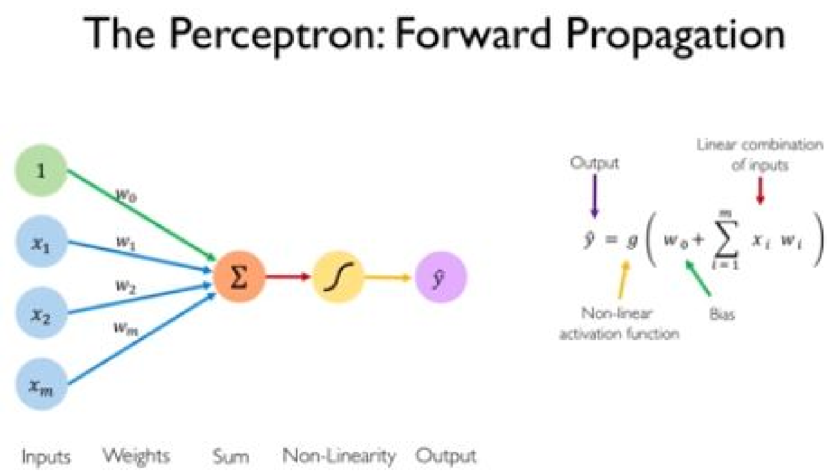
INTRO TO DEEP LEARNING

Learning from the things are working later ~ learning features from underlying features.

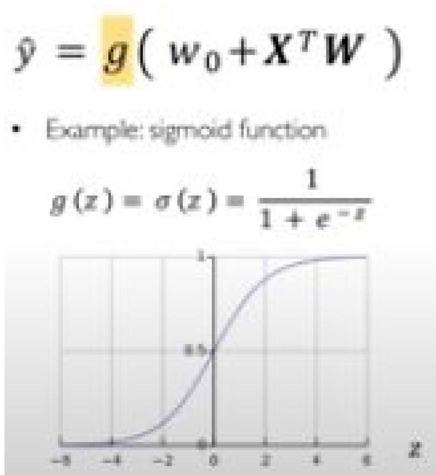
Big Data -> Hardware -> Software

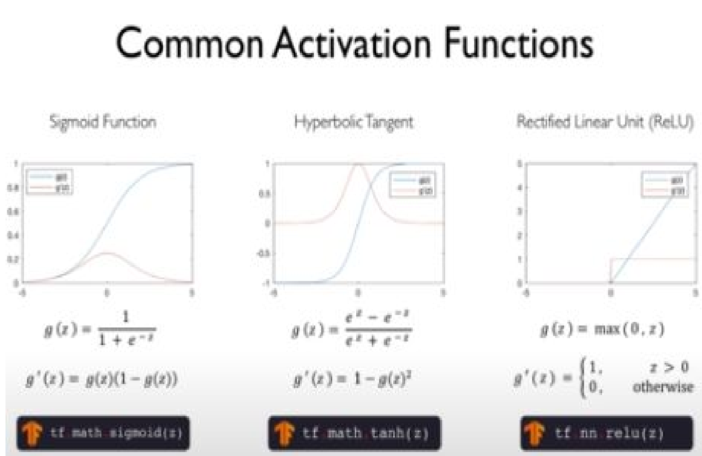
Perceptron :



Activation Function :

Takes a real number in x axis and converts it into a number in 0 and 1.





Purpose of AF is - to introduce non-linearities in functions.

Hidden layers , dense layer

Quantifying Loss : measures the cost incurred from incorrect predictions.

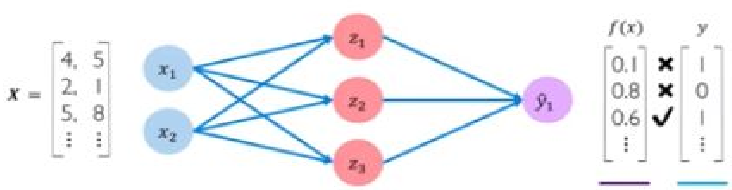
$$\mathcal{L}(\underbrace{f(x^{(i)}; \boldsymbol{W})}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

Empirical Loss : total loss over the entire dataset.

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(\underbrace{f(x^{(i)}; \boldsymbol{W})}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

Binary Cross Entropy Loss

Cross entropy loss *can be used with models that output a probability between 0 and 1*

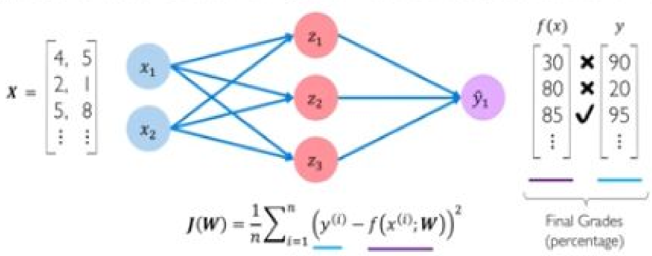


$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\left(\underbrace{f(x^{(i)}; \boldsymbol{W})}_{\text{Predicted}}\right) + \underbrace{(1 - y^{(i)})}_{\text{Actual}} \log\left(1 - \underbrace{f(x^{(i)}; \boldsymbol{W})}_{\text{Predicted}}\right)$$

```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(y, predicted) )
```

Mean Squared Error Loss

Mean squared error loss *can be used with regression models that output continuous real numbers*



$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \left(\underbrace{y^{(i)}}_{\text{Actual}} - \underbrace{f(x^{(i)}; \boldsymbol{W})}_{\text{Predicted}}\right)^2$$

```
loss = tf.reduce_mean( tf.square(tf.subtract(y, predicted)) )
```

Loss Optimisation : finding the network weights that achieve the lowest loss through maybe gradient descent.

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

```
import tensorflow as tf

weights = tf.Variable([tf.random.normal()])

while True:   # loop forever
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
        gradient = g.gradient(loss, weights)

    weights = weights - lr * gradient
```

Backpropagation : chain rule ~

Optimization through gradient descent

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \boxed{\eta} \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$$

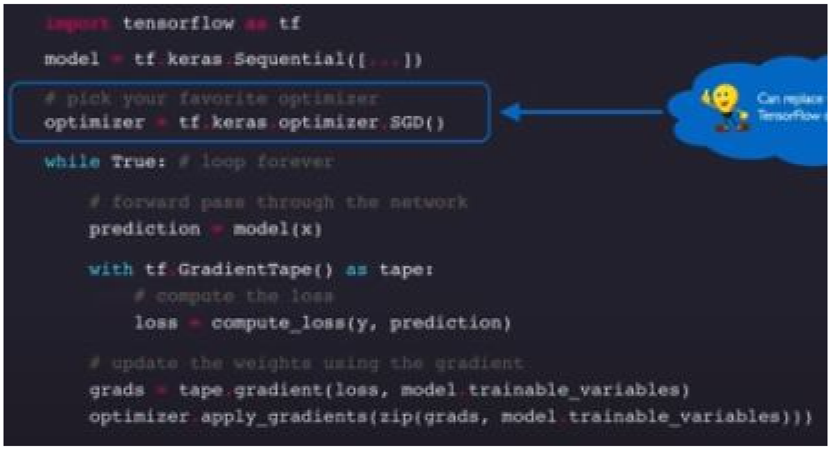
How can we set the learning rate?

Adaptive Learning Rates ~

- Learning rates are no longer fixed
- Can be made larger or smaller depending on:
  - how large gradient is
  - how fast learning is happening
  - size of particular weights
  - etc...

Gradient Descent Algorithms

Algorithm	TF Implementation	Refe
SGD	tf.keras.optimizers.SGD	Kiefer & Wolfowitz. "Sto Maximum of a Regressi
Adam	tf.keras.optimizers.Adam	Kingma et al. "Adam: A I Optimization." 2014.
Adadelta	tf.keras.optimizers.Adadelta	Zeiler et al. "ADADELT. Method." 2012.
Adagrad	tf.keras.optimizers.Adagrad	Duchi et al. "Adaptive S Learning and Stochastic
RMSProp	tf.keras.optimizers.RMSProp	



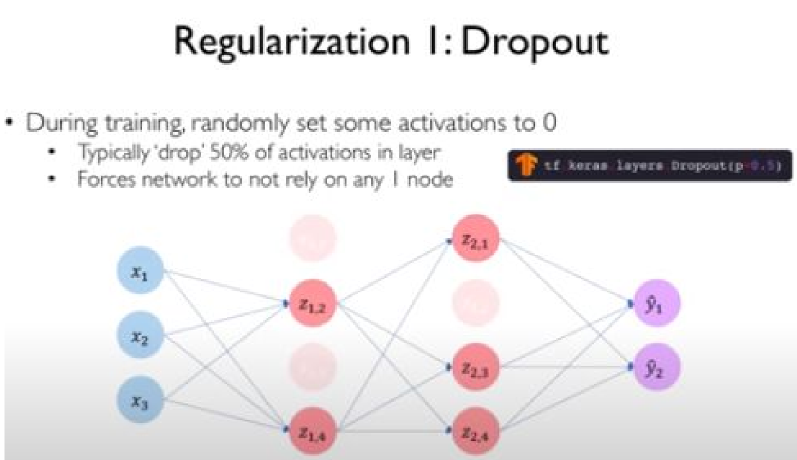
Stochastic gradient descent ~ means using a single point.

Using a single point for gradient descent in very noisy and using all the data points is very computationally expensive , therefore the solution is ~

Using mini batches of points ~ and true is calculated by taking the average , therefore smoother convergence and allows greater learning rates , enables parallelism therefore faster.

Overfitting exists so to deal with it ~

Regularisation ~ technique that constraints our optimisation problem to discourage complex models ~ improve generalisation.



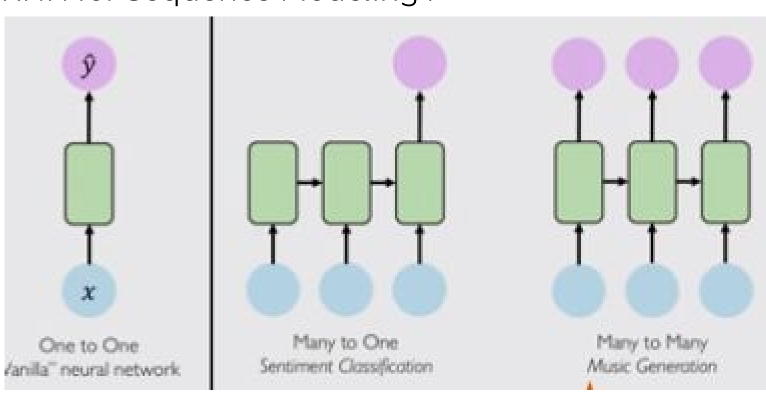
RECURRENT NEURAL NETWORKS

Sequential Modelling : predict the next thing that will happen after a previous states are given

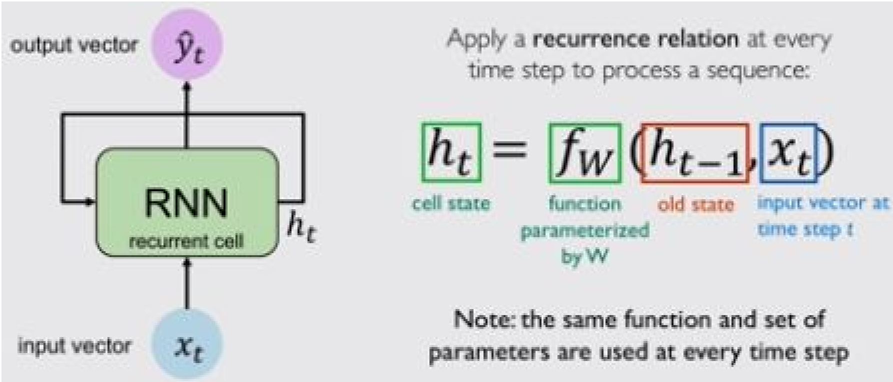
If word prediction then if using BagOfWords then the order is not reserved therefore Separate parameters should be used to encode the sentence.

In standard Feed Forward -> it goes one to one passing of data

RNN for Sequence Modeling :



RNNs have a loop in them that enables them to maintain an internal state , not like a vanilla NN where the final output is only present.



Output Vector

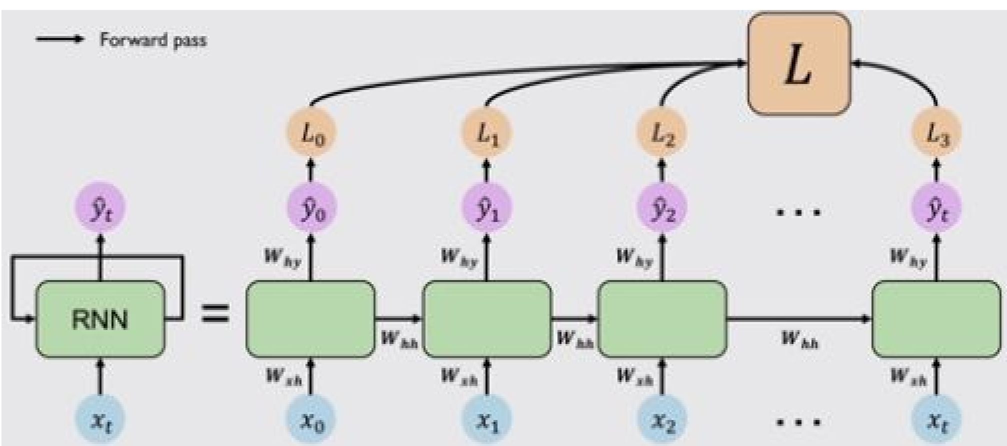
$$\hat{y}_t = \boldsymbol{W}_{hy}^T h_t$$

Update Hidden State

$$h_t = \tanh(\boldsymbol{W}_{hh}^T h_{t-1} + \boldsymbol{W}_{xh}^T x_t)$$

Input Vector

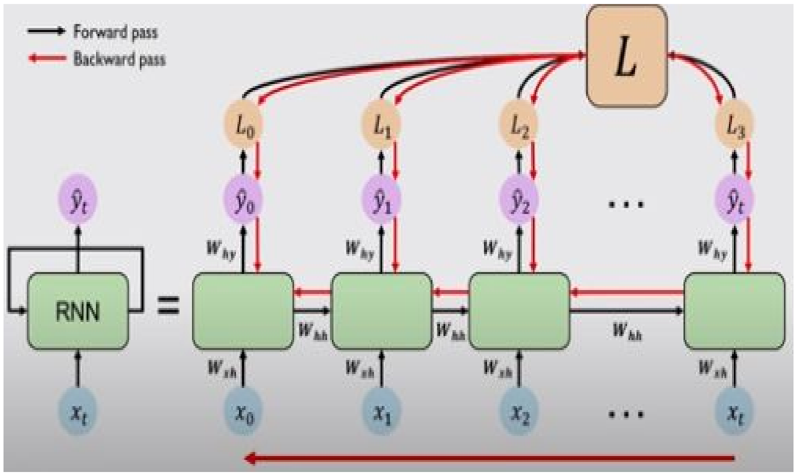
$$x_t$$



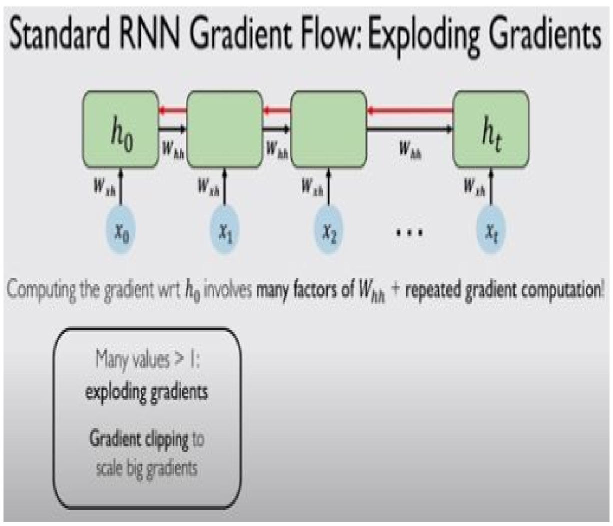

```
tf.keras.layers.SimpleRNN(rnn_units)
```

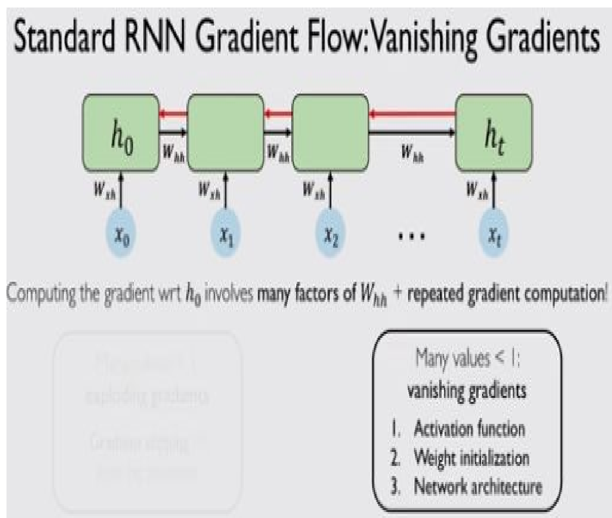
Backpropagation through time :

In RNNs at each time backprop is done also in total ~



Standard RNN Gradient Flow can have 2 problems :

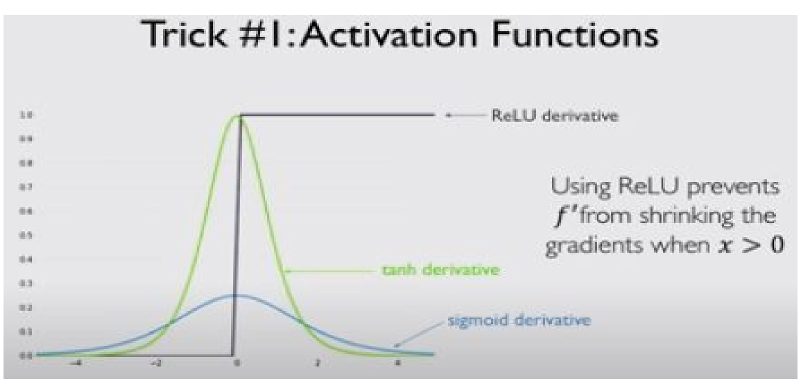




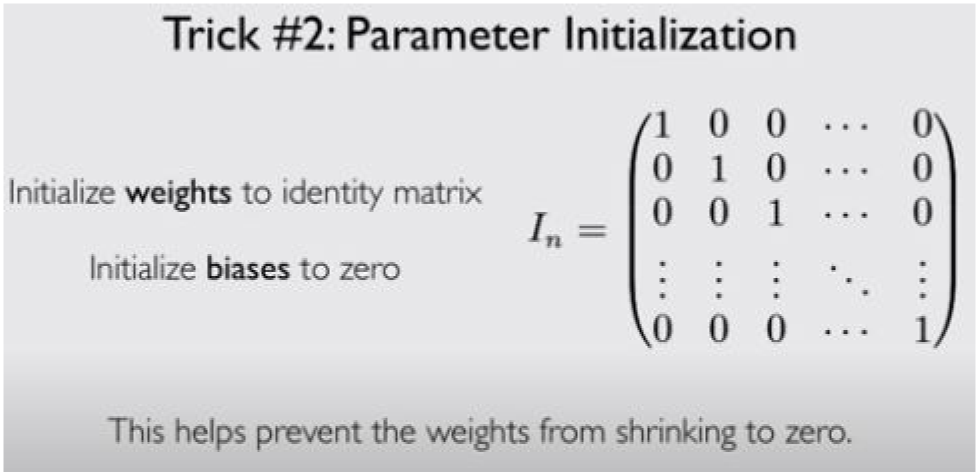
The problem of LongTermDependencies (vanishing gradient)~

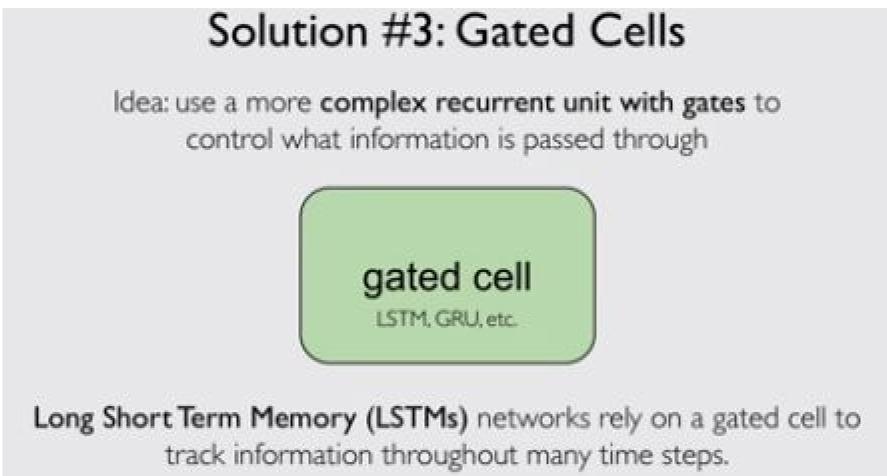
Multiply many small numbers together -> errors which are in further back time steps keep having smaller and smaller gradients -> this bias es parameters to capture dependencies in models , thus standard RNNs becoming less capable.

To solve the above problem :



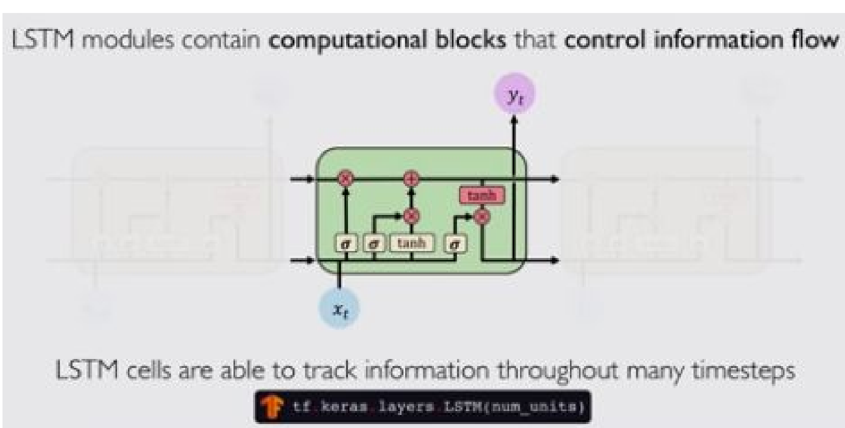
Both tanh and sigmoid have derivatives less than 1.
Therefore use RELU but x>0 only then.



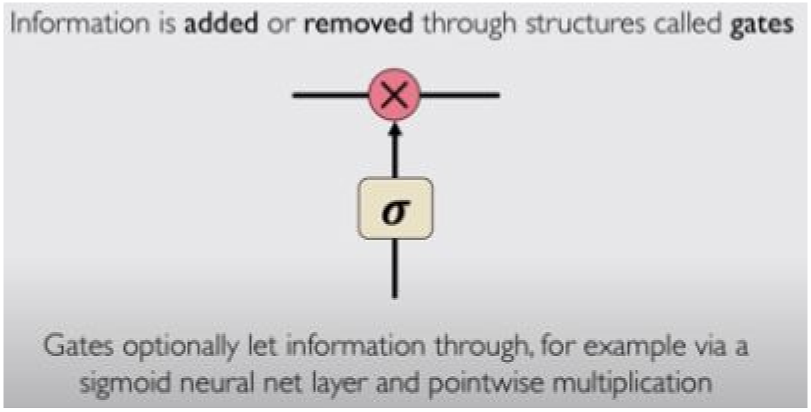


The above is well suited for learning tasks.

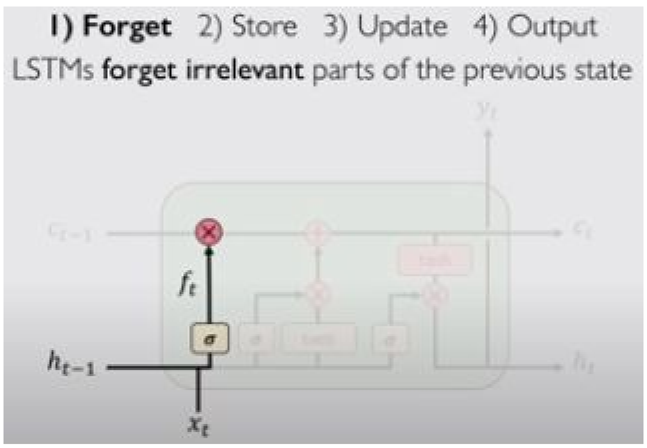
Long Short Term Memory (LSTM) Networks :

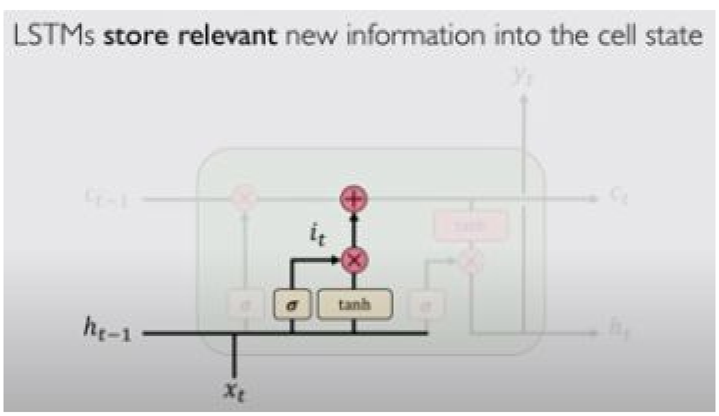


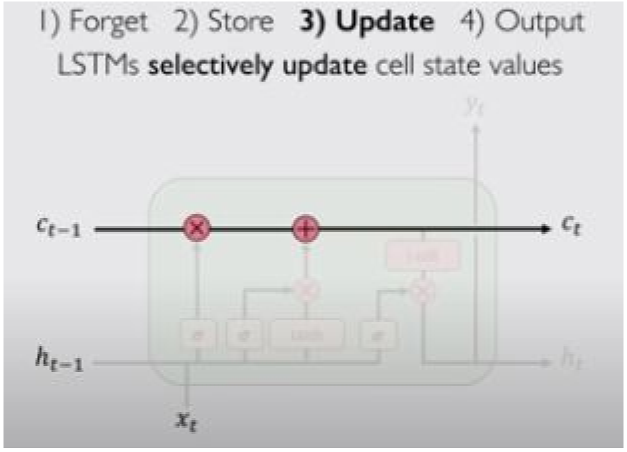
Gates are an important part in LSTMs

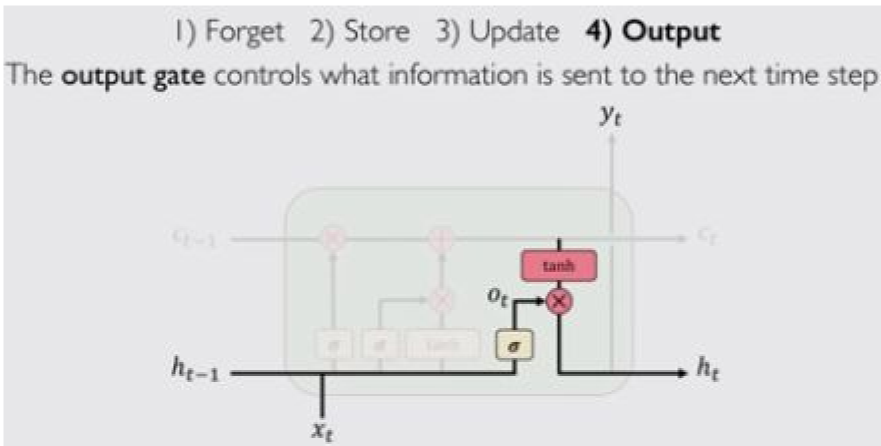


LSTMs work in 4 steps :

1. 

2. 

3. 

4. 

All these allow LSTMs to have an uninterrupted gradient flow.

RNN Applications :

Example :

Music Generation ~ at each level the next tune is predicted.
Sentiment Classification ~
Machine Translation ~ encoding bottleneck is a problem so attention mechanism is used ~ all states of time steps are accessed and trained upon individually.
Trajectory Prediction ~ Self Driving Car
Environmental Modelling