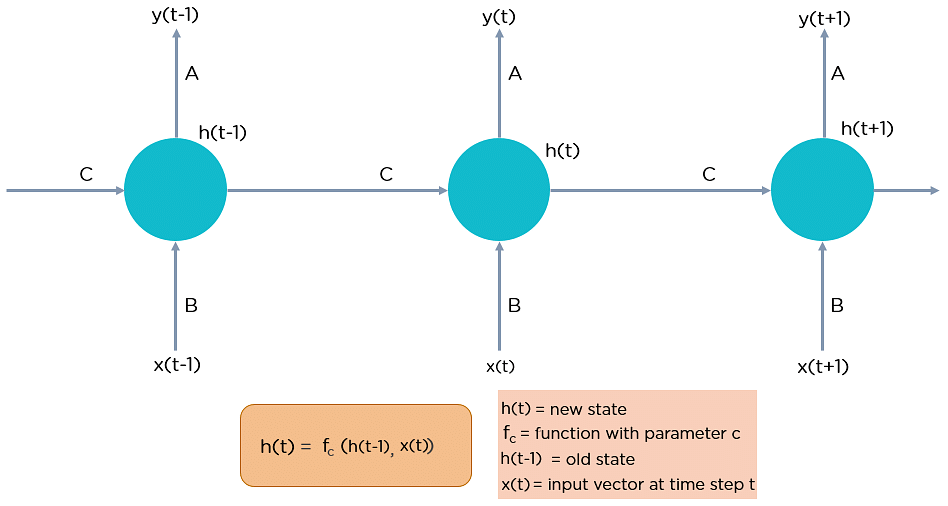
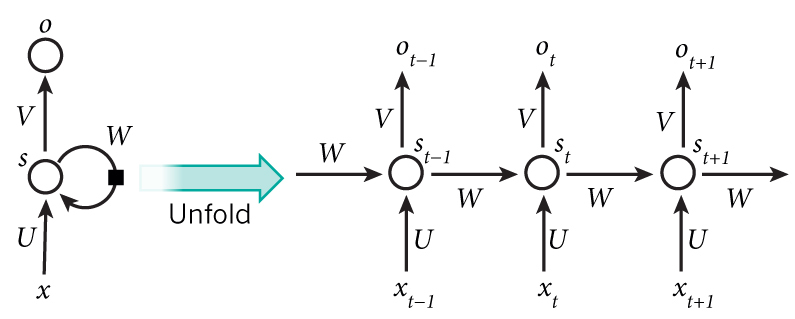
**RNN**

RNN works on the principle of saving the output of a particular layer and feeding this back to the input in order to predict the output of the layer.

An RNN can handle sequential data, accepting the current input data, and previously received inputs. RNNs can memorize previous inputs due to their internal memory.





In Recurrent Neural networks, the information cycles through a loop to the middle hidden layer.

The input layer ‘x’ takes in the input to the neural network and processes it and passes it onto the middle layer.

In Normal neural network The middle layer ‘h’ can consist of multiple hidden layers, each with its own activation functions and weights and biases. But The Recurrent Neural Network will standardize the different activation functions and weights and biases so that each hidden layer has the same parameters. Then, instead of creating multiple hidden layers, it will create one and loop over it as many times as required.

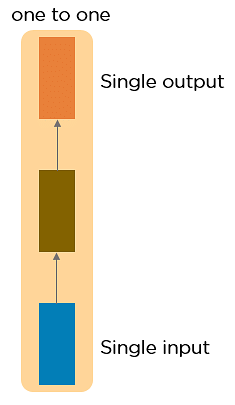
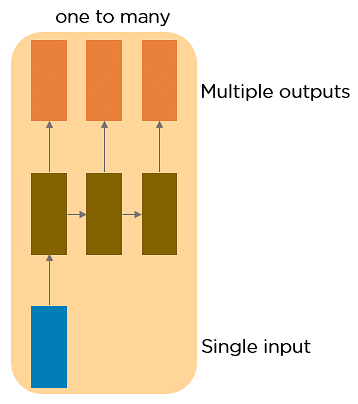
Types of Recurrent Neural Networks

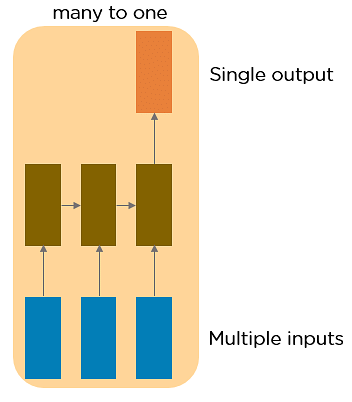
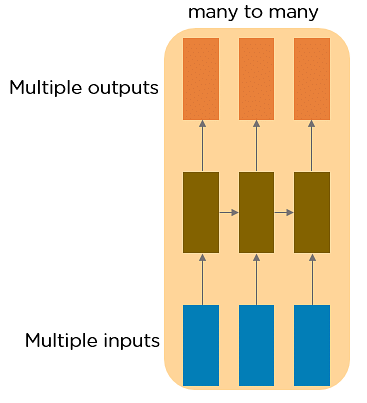
There are four types of Recurrent Neural Networks:

One to One One to Many

Many to One Many to Many

1. A single time step of the input is supplied to the network i.e. xt is supplied to the network
2. We then calculate its current state using a combination of the current input and the previous state i.e. we calculate ht
3. The current ht becomes ht-1 for the next time step
4. We can go as many time steps as the problem demands and combine the information from all the previous states
5. Once all the time steps are completed the final current state is used to calculate the output yt
6. The output is then compared to the actual output and the error is generated
7. The error is then backpropagated to the network to update the weights(we shall go into the details of backpropagation in further sections) and the network is trained.





## Long Short-Term Memory Networks

LSTMs are a special kind of RNN — capable of learning long-term dependencies by remembering information for long periods is the default behavior.

LSTMs make small modifications to the information by multiplications and additions. With LSTMs, the information flows through a mechanism known as cell states. This way, LSTMs can selectively remember or forget things. The information at a particular cell state has three different dependencies.

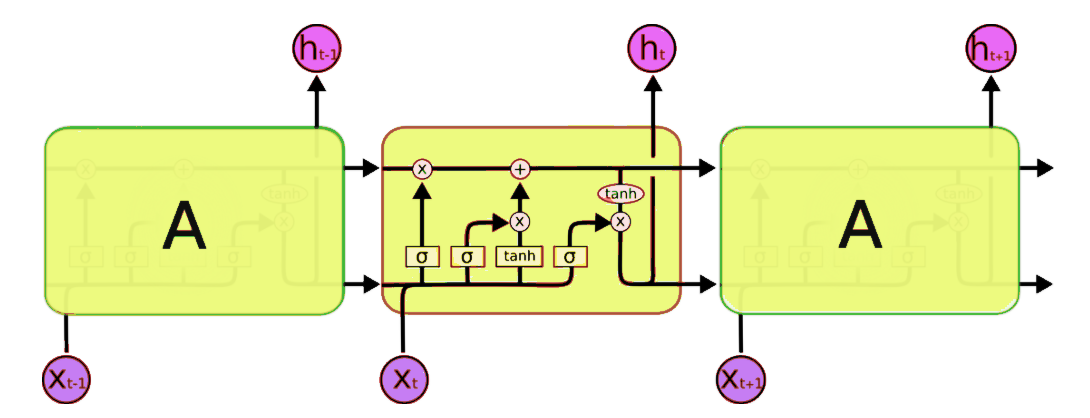
These dependencies can be generalized to any problem as:

1. The previous cell state *(i.e. the information that was present in the memory after the previous time step)*
2. The previous hidden state *(i.e. this is the same as the output of the previous cell)*
3. The input at the current time step *(i.e. the new information that is being fed in at that moment)*

Another important feature of LSTM is its analogy with conveyor belts!

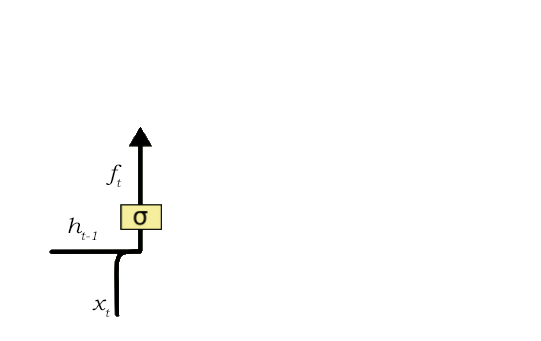
Just because of this property of LSTMs, where they do not manipulate the entire information but rather modify them slightly, they are able to *forget*and *remember*things selectively.

## Architecture of LSTMs



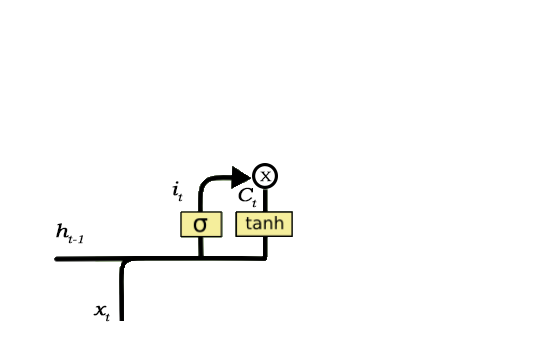
A typical LSTM network is comprised of different memory blocks called cells. here are two states that are being transferred to the next cell; the cell state and the hidden state. The memory blocks are responsible for remembering things and manipulations to this memory is done through three major mechanisms, called gates.

**Forget Gate**



* A forget gate is responsible for removing information from the cell state.
* The information that is no longer required for the LSTM to understand things or the information that is of less importance is removed via multiplication of a filter. This is required for optimizing the performance of the LSTM network.
* This gate takes in two inputs; h\_t-1 and x\_t.
* h\_t-1 is the hidden state from the previous cell or the output of the previous cell and x\_t is the input at that particular time step.
* The given inputs are multiplied by the weight matrices and a bias is added. Following this, the sigmoid function is applied to this value.
* The sigmoid function outputs a vector, with values ranging from 0 to 1, corresponding to each number in the cell state.
* Basically, the sigmoid function is responsible for deciding which values to keep and which to discard.
* If a ‘0’ is output for a particular value in the cell state, it means that the forget gate wants the cell state to forget that piece of information completely.
* Similarly, a ‘1’ means that the forget gate wants to remember that entire piece of information. This vector output from the sigmoid function is multiplied to the cell state.

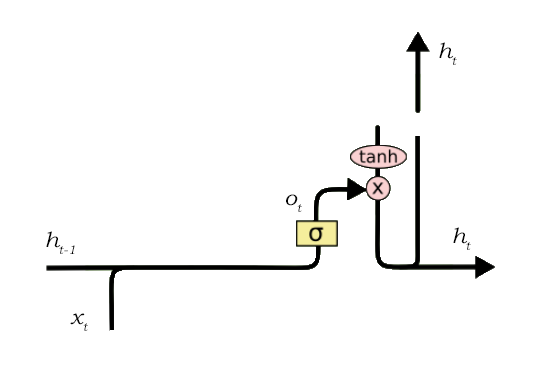
**Input Gate**



The input gate is responsible for the addition of information to the cell state. This addition of information is basically three-step process as seen from the diagram above.

1. Regulating what values need to be added to the cell state by involving a sigmoid function. This is basically very similar to the forget gate and acts as a filter for all the information from h\_t-1 and x\_t.
2. Creating a vector containing all possible values that can be added (as perceived from h\_t-1 and x\_t) to the cell state. This is done using the **tanh**function, which outputs values from -1 to +1.
3. Multiplying the value of the regulatory filter (the sigmoid gate) to the created vector (the tanh function) and then adding this useful information to the cell state via addition operation.

**Output Gate**

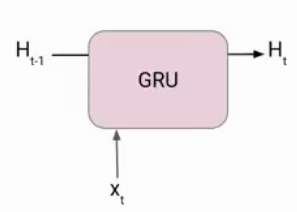


This job of selecting useful information from the current cell state and showing it out as an output is done via the output gate.

The functioning of an output gate can again be broken down to three steps:

1. Creating a vector after applying **tanh**function to the cell state, thereby scaling the values to the range -1 to +1.
2. Making a filter using the values of h\_t-1 and x\_t, such that it can regulate the values that need to be output from the vector created above. This filter again employs a sigmoid function.
3. Multiplying the value of this regulatory filter to the vector created in step 1, and sending it out as a output and also to the hidden state of the next cell.

**GRU**

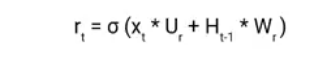


At each timestamp t, it takes an input Xt and the hidden state Ht-1 from the previous timestamp t-1. Later it outputs a new hidden state Ht which again passed to the next timestamp.

Now there are primarily two gates in a GRU as opposed to three gates in an LSTM cell. The first gate is the Reset gate and the other one is the update gate.

**Reset Gate (Short term memory)**

The Reset Gate is responsible for the short-term memory of the network i.e the hidden state (Ht). Here is the equation of the Reset gate.



If you remember from the LSTM gate equation it is very similar to that. The value of **rt**will range from 0 to 1 because of the sigmoid function. Here Ur and Wr are weight matrices for the reset gate.

**Update Gate (Long Term memory)**

Similarly, we have an Update gate for long-term memory and the equation of the gate is shown below.

Gated recurrent unit - Update Gate (Long Term memory)

The only difference is of weight metrics i.e Uu and Wu.

**Image Segmentation**

We can divide or partition the image into various parts called segments. It’s not a great idea to process the entire image at the same time as there will be regions in the image which do not contain any information. By dividing the image into segments, we can make use of the important segments for processing the image. That, in a nutshell, is how image segmentation works.

An image is a collection or set of different pixels. We group together the pixels that have similar attributes using image segmentation.

Object detection builds a bounding box corresponding to each class in the image. But it tells us nothing about the shape of the object. We only get the set of bounding box coordinates. We want to get more information – this is too vague for our purposes.

Image segmentation creates a pixel-wise mask for each object in the image. This technique gives us a far more granular understanding of the object(s) in the image.

We can broadly divide image segmentation techniques into two types.

* Semantic
* Instance based

1. Region-based Segmentation
2. Edge Detection Segmentation
3. Image Segmentation based on Clustering
4. Mask R-CNN

**AlexNet Architecture:**

The AlexNet architecture consists of five convolutional layers, some of which are followed by maximum pooling layers and then three fully-connected layers and finally a 1000-way softmax classifier.

**First Layer:**

The input for AlexNet is a 227x227x3 RGB image which passes through the first convolutional layer with 96 feature maps or filters having size 11×11 and a stride of 4. The image dimensions changes to 55x55x96.

Then the AlexNet applies maximum pooling layer or sub-sampling layer with a filter size 3×3 and a stride of two. The resulting image dimensions will be reduced to 27x27x96.

**Second Layer :**

Next, there is a second convolutional layer with 256 feature maps having size 5×5 and a stride of 1.

Then there is again a maximum pooling layer with filter size 3×3 and a stride of 2. This layer is same as the second layer except it has 256 feature maps so the output will be reduced to 13x13x256.

**Third, Fourth and Fifth Layers:**

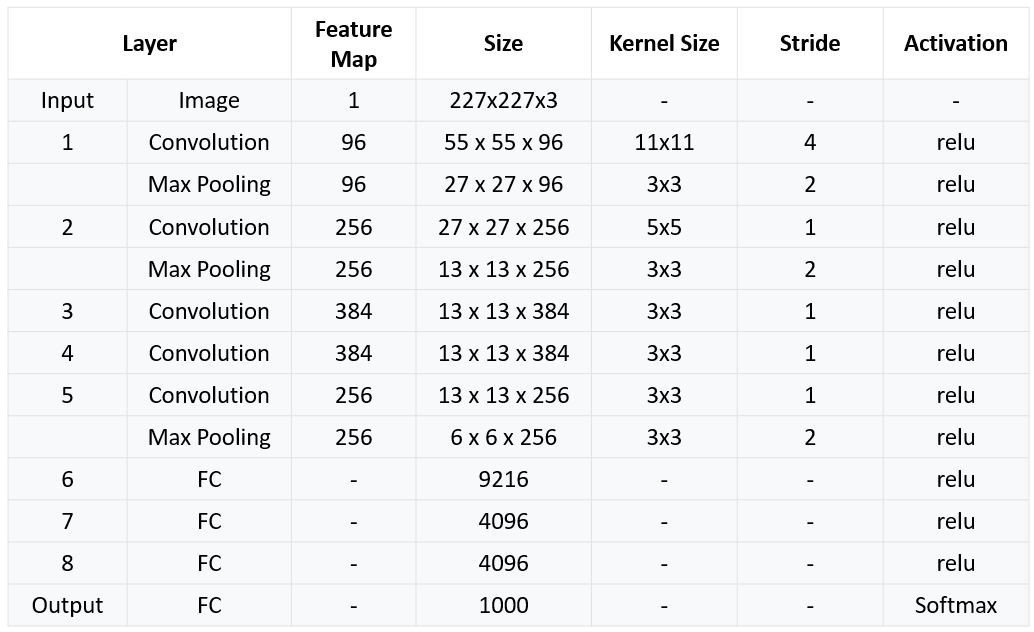
The third, fourth and fifth layers are convolutional layers with filter size 3×3 and a stride of one. The first two used 384 feature maps where the third used 256 filters.

The three convolutional layers are followed by a maximum pooling layer with filter size 3×3, a stride of 2 and have 256 feature maps.

**Sixth Layer:**  
The convolutional layer output is flattened through a fully connected layer with 9216 feature maps each of size 1×1.

**Seventh and Eighth Layers:**  
Next is again two fully connected layers with 4096 units.

**Output Layer:**  
Finally, there is a softmax output layer ŷ with 1000 possible values.



**Program**

import keras  
from keras.models import Sequential  
from keras.layers import Dense, Activation, Dropout, Flatten, Conv2D, MaxPooling2D  
from keras.layers.normalization import BatchNormalization  
import numpy as np  
np.random.seed(1000)

#Instantiate an empty model  
model = Sequential()

# 1st Convolutional Layer  
model.add(Conv2D(filters=96, input\_shape=(224,224,3), kernel\_size=(11,11), strides=(4,4), padding=’valid’))  
model.add(Activation(‘relu’))  
# Max Pooling  
model.add(MaxPooling2D(pool\_size=(2,2), strides=(2,2), padding=’valid’))

# 2nd Convolutional Layer  
model.add(Conv2D(filters=256, kernel\_size=(11,11), strides=(1,1), padding=’valid’))  
model.add(Activation(‘relu’))  
# Max Pooling  
model.add(MaxPooling2D(pool\_size=(2,2), strides=(2,2), padding=’valid’))

# 3rd Convolutional Layer  
model.add(Conv2D(filters=384, kernel\_size=(3,3), strides=(1,1), padding=’valid’))  
model.add(Activation(‘relu’))

# 4th Convolutional Layer  
model.add(Conv2D(filters=384, kernel\_size=(3,3), strides=(1,1), padding=’valid’))  
model.add(Activation(‘relu’))

# 5th Convolutional Layer  
model.add(Conv2D(filters=256, kernel\_size=(3,3), strides=(1,1), padding=’valid’))  
model.add(Activation(‘relu’))  
# Max Pooling  
model.add(MaxPooling2D(pool\_size=(2,2), strides=(2,2), padding=’valid’))

# Passing it to a Fully Connected layer  
model.add(Flatten())  
# 1st Fully Connected Layer  
model.add(Dense(4096, input\_shape=(224\*224\*3,)))  
model.add(Activation(‘relu’))  
# Add Dropout to prevent overfitting  
model.add(Dropout(0.4))

# 2nd Fully Connected Layer  
model.add(Dense(4096))  
model.add(Activation(‘relu’))  
# Add Dropout  
model.add(Dropout(0.4))

# 3rd Fully Connected Layer  
model.add(Dense(1000))  
model.add(Activation(‘relu’))  
# Add Dropout  
model.add(Dropout(0.4))

# Output Layer  
model.add(Dense(17))  
model.add(Activation(‘softmax’))

model.summary()

# Compile the model  
model.compile(loss=keras.losses.categorical\_crossentropy, optimizer=’adam’, metrics=[“accuracy”])