**Problem Statement**

Imagine you are working as a data scientist at a home electronics company which manufactures state of the art **smart televisions**. You want to develop a cool feature in the smart-TV that can **recognise five different gestures** performed by the user which will help users control the TV without using a remote.

The gestures are continuously monitored by the webcam mounted on the TV. Each gesture corresponds to a specific command:

* Thumbs up:  Increase the volume
* Thumbs down: Decrease the volume
* Left swipe: 'Jump' backwards 10 seconds
* Right swipe: 'Jump' forward 10 seconds
* Stop: Pause the movie

 Each video is a sequence of 30 frames (or images).

## ****Understanding the Dataset****

The training data consists of a few hundred videos categorised into one of the five classes. Each video (typically 2-3 seconds long) is divided into a **sequence of 30 frames(images)**. These videos have been recorded by various people performing one of the five gestures in front of a webcam - similar to what the smart TV will use.

 The data is in a zip file. The zip file contains a 'train' and a 'val' folder with two CSV files for the two folders. These folders are in turn divided into subfolders where each subfolder represents a video of a particular gesture. Each subfolder, i.e. a video, contains 30 frames (or images). Note that all images in a particular video subfolder have the same dimensions but different videos may have different dimensions. Specifically, videos have two types of dimensions - either 360x360 or 120x160 (depending on the webcam used to record the videos). Hence, you will need to do some pre-processing to standardise the videos.

 Each row of the CSV file represents one video and contains three main pieces of information - the name of the subfolder containing the 30 images of the video, the name of the gesture and the numeric label (between 0-4) of the video.

 Your task is to train a model on the 'train' folder which performs well on the 'val' folder as well (as usually done in ML projects). We have withheld the test folder for evaluation purposes - your final model's performance will be tested on the 'test' set.

 To get started with the model building process, you first need to get the data on your storage.

In order to get the data on the storage, perform the following steps in order

1. Open the terminal
2. g0 down <https://drive.google.com/uc?id=1ehyrYBQ5rbQQe6yL4XbLWe3FMvuVUGiL>
3. unzip Project\_data.zip

 Now that you have got the data on the storage, let's look at the different choices of architectures you can use.

# Two Architectures: 3D Convs and CNN-RNN Stack

After understanding and acquiring the dataset, the next step is to try out different architectures to solve this problem.

 For analysing videos using neural networks, **two types of architectures**are used commonly. One is the standard **CNN + RNN architecture** in which you pass the images of a video through a CNN which extracts a feature vector for each image, and then pass the sequence of these feature vectors through an RNN. This is something you are already familiar with (in theory).

 The other popular architecture used to process videos is a natural extension of CNNs - a **3D convolutional network**. In this project, you will try both these architectures.

Thus, there are two types of architecture commonly used for analysing videos, both explained below.

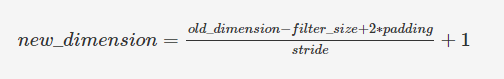
**Convolutions + RNN**  
The conv2D network will extract a feature vector for each image, and a sequence of these feature vectors is then fed to an RNN-based network. The output of the RNN is a regular softmax (for a classification problem such as this one).

**3D Convolutional Network, or Conv3D**  
3D convolutions are a natural extension to the 2D convolutions you are already familiar with. Just like in 2D conv, you move the filter in two directions (x and y), in 3D conv, you move the filter in three directions (x, y and z). In this case, the input to a 3D conv is a video (which is a sequence of 30 RGB images). If we assume that the shape of each image is 100x100x3, for example, the video becomes a 4-D tensor of shape 100x100x3x30 which can be written as (100x100x30)x3 where 3 is the number of channels. Hence, deriving the analogy from 2-D convolutions where a 2-D kernel/filter (a square filter) is represented as (fxf)xc where f is filter size and c is the number of channels, a 3-D kernel/filter (a 'cubic' filter) is represented as (fxfxf)xc (here c = 3 since the input images have three channels). This cubic filter will now '3D-convolve' on each of the three channels of the (100x100x30) tensor.

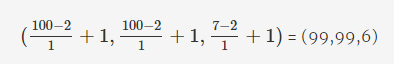
As an example, let's calculate the output shape and the number of parameters in a Conv3D with an example of a video having 7 frames. Each image is an RGB image of dimension 100x100x3. Here, the number of channels is 3.

The input (video) is then 7 images stacked on top of each other, so the shape of the input is (100x100x7)x3, i.e (length x width x number of images) x number of channels. Now, let's use a 3-D filter of size 2x2x2. This is represented as (2x2x2)x3 since the filter has the same number of channels as the input (exactly like in 2D convs).

Now let's perform a 3D convolution using a (2x2x2) filter on a (100x100x7) matrix (without any padding and stride = 1). You know that the **output size after convolution** is given by:



In 3D convolutions, the filter convolves in three directions (exactly like it does in two dimensions in 2D convs), so you can extend this formula to the third dimension as well. You get the output shape as:



Thus, the output shape after performing a 3D conv with one filter is (99x99x6). Now, if we do (say) 24 such 3D convolutions, the output shape will become (99x99x6)x24. Hence, the new number of channels for the next Conv3D layer becomes 24. This is very similar to what happens in conv2D.

Now let's calculate **the number of trainable parameters** if the input shape is (100x100x7)x3 and it is convolved with 24 3D filters of size (2x2x2) each, expressed as (2x2x2)x3 to give an output of shape (99x99x6)x24. Each filter will have 2x2x2 = 8 trainable parameters for **each** **of the 3 channels**. Also, here we consider that there is **one** **bias per filter**. Hence, each filter has 8x3 + 1  = 25 trainable parameters. For 24 such filters, we get 25x24 = 600 trainable parameters.

 There are a few key things to note about the conv-RNN architecture:

1. You can use **transfer learning** in the 2D CNN layer rather than training your own CNN
2. GRU can be a better choice than an LSTM since it has lesser number of gates (and thus parameters)