Long-term Recurrent Convolutional Networks (LRCNs) for Activity Recognition

Ayush Pandey (2K20/SE/38) , B.Tech , Department of Software Engineering

Abstract

Models based on deep convolutional networks have dominated recent image interpretation tasks; we implemented a combination of CNN and Long-Term Short Memory and used it for the activity recognition. We describe a class of recurrent convolutional architectures which is end-to-end trainable and suitable for large-scale visual understanding tasks, and demonstrate the value of these models for activity recognition . In contrast to previous models which assume a fixed visual representation or perform simple temporal averaging for sequential processing, recurrent convolutional models are "doubly deep" in that they learn compositional representations in space and time. Learning long-term dependencies is possible when nonlinearities are incorporated into the network state updates. Differentiable recurrent models are appealing in that they can directly map variable-length inputs (e.g., videos) to variable-length outputs (e.g., natural language text) and can model complex temporal dynamics; yet they can be optimized with backpropagation. Our recurrent sequence models are directly connected to modern visual convolutional network models and can be jointly trained to learn temporal dynamics and convolutional perceptual representations.

1. INTRODUCTION

With the development of various models in machine learning, we have already achieved the ability to predict things by figuring out the patterns from the previous events. In recent years, we applied these concepts to images , videos , sequence of the texts etc and they performed good but not perfectly. With the further development in the field of Deep Learning, various models like Artificial Neural Network (ANN), Convolutional Neural Network (CNN) etc have been proposed , which are better at dealing with sequential data, images etc. Convolutional Neural Network (CNN) is one of the best methods for image classification. In this paper, we implemented a combination of Convolutional Neural Network (CNN) and Long-Short Term Memory and trained the model with different videos for identifying the human actions in them. This combination of CNN and LSTM is known as Long-Term Recurrent Convolutional Network (LRCNs).

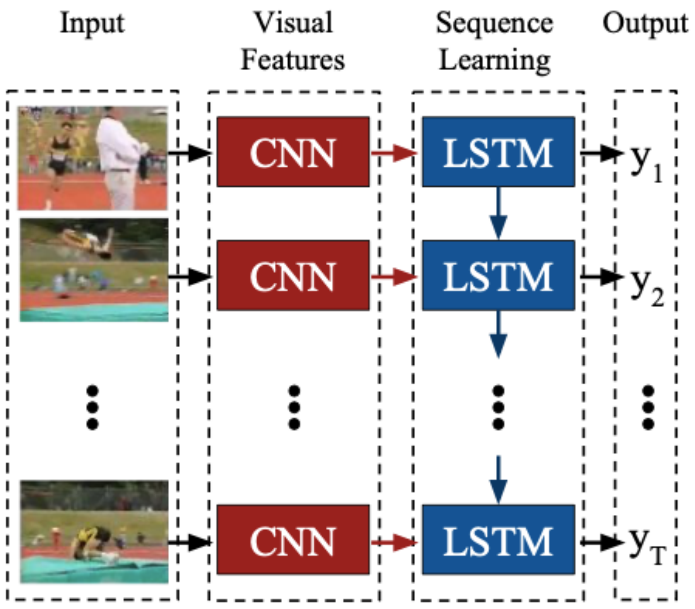


Fig: Long-Term Recurrent Neural Network

Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other.

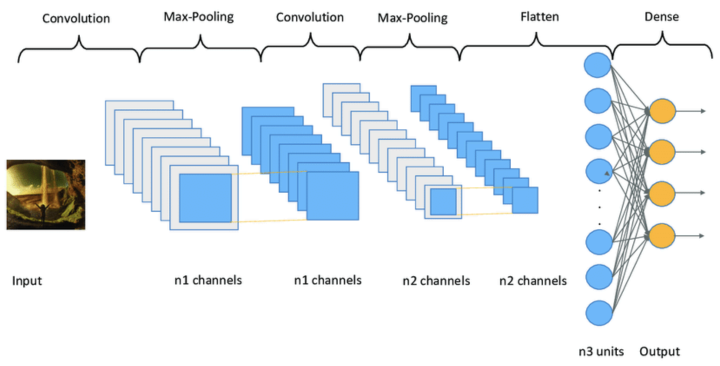


Fig: Convolutional Neural Network

Recurrent Neural Network(RNN) are a type of <u>Neural Network</u> where the **output from previous step are fed as input to the current step**. In traditional neural networks, all the inputs and outputs are independent of each other, but in cases like when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is **Hidden state**, which remembers some information about a sequence.

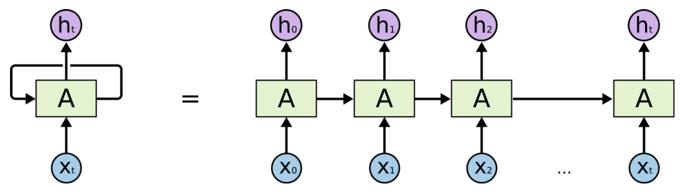


Fig: Recurrent Neural Network

Long Short Term Memory is a type of RNN ,which is able to memorise the long-term dependencies. It is the special case of RNN, where the model stores the required data over the number of steps and is used to predict the future values , texts etc.

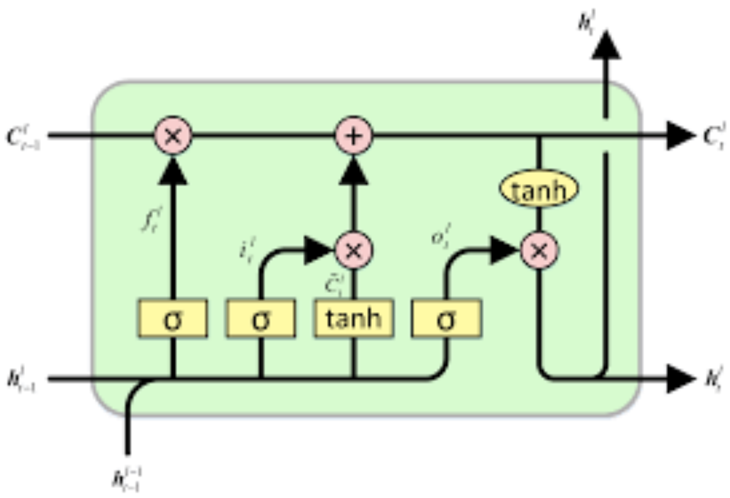


Fig: Long Short Term Memory

2. Background

2.1 Artificial Neural Network

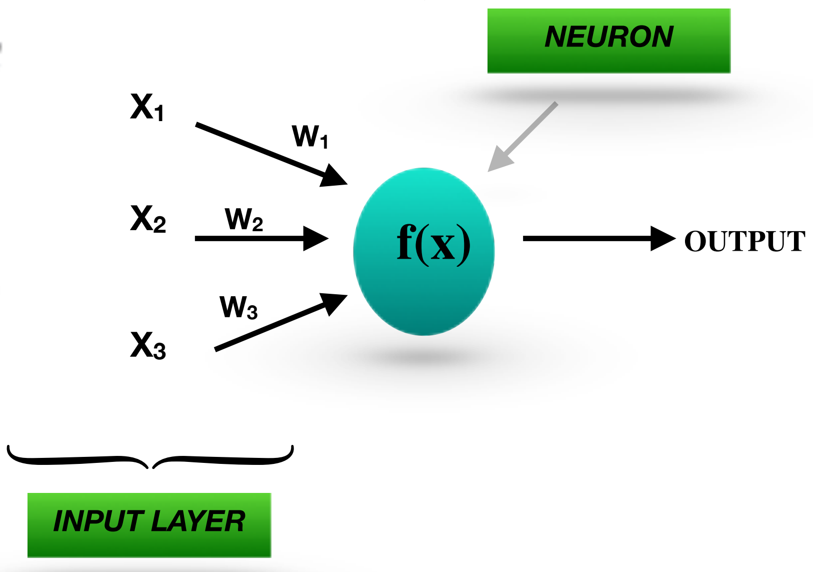
Artificial Neural Network is a series of algorithms that are trying to mimic the human brain and find the relationship between the sets of data. It is being used in various use-cases like in regression, classification, Image Recognition and many more. In this section, we will see the various components of the ANN and their functions.

2.1.1 Components of ANN

✦ **Neurons (Perceptron)**

The single unit of the Artificial Neural Network where the processing of the given inputs takes place is known as Neuron or Perceptron.

It actually mimics the single neuron unit in the Human nervous system. It is responsible for taking decisions individually.

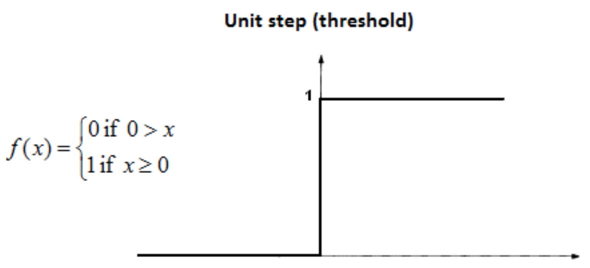


F(x) is known as Activation Function, which is denoted by,

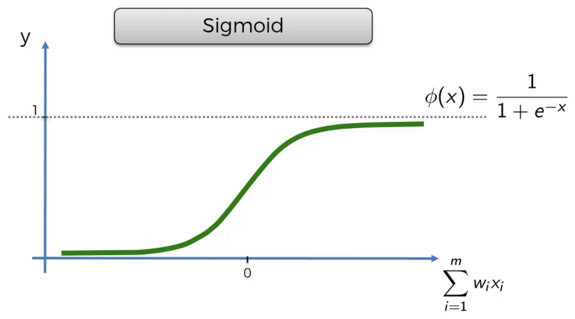
$$f(x) = \phi\left(\sum_{i=1}^{m} w_i x_i\right)$$

There are different types of activation functions . Some of them are listed below:

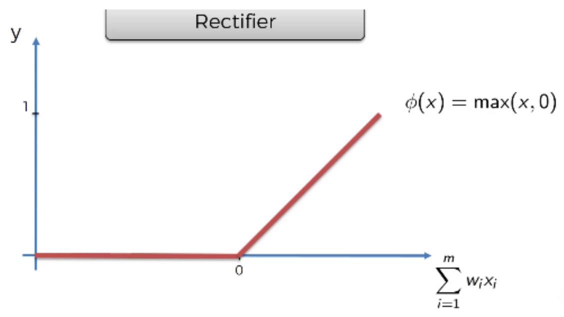
- Threshold Function



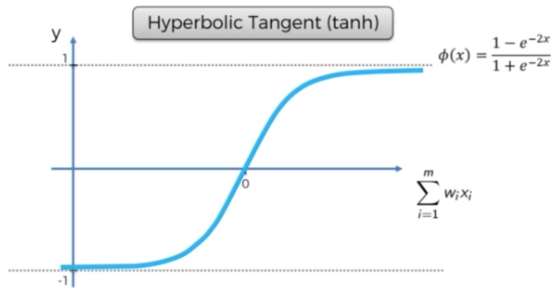
- Sigmoid Function



- Rectifier Linear Unit (ReLU)

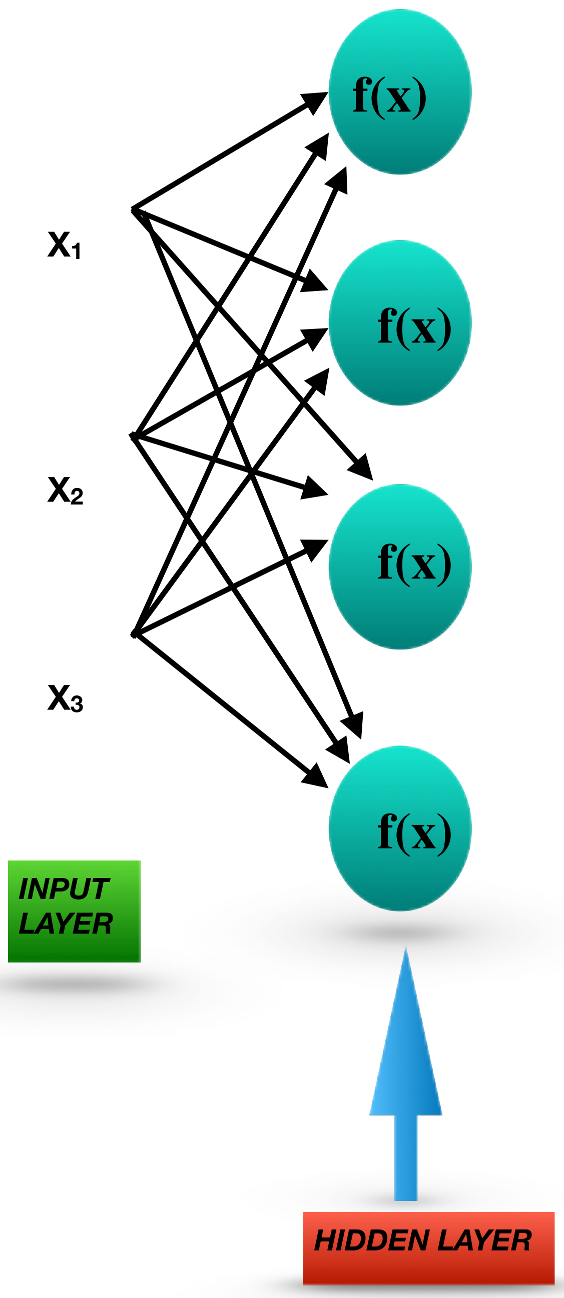


- Hyperbolic Tangent



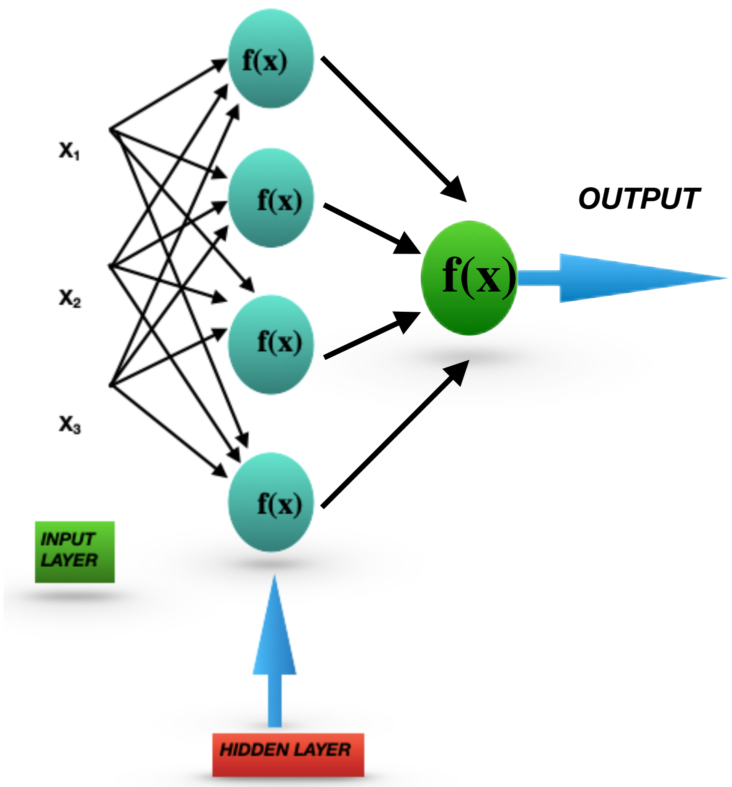
✦ Hidden Layer

It is the series of the neurones between the input and the output layer , which processes the inputs provided and accordingly gives the output.



Each of the neurone in the hidden layer performs the activation function *f(x)* on the inputs. Usually, Rectifier Linear Unit (ReLU) is used in the hidden layer.

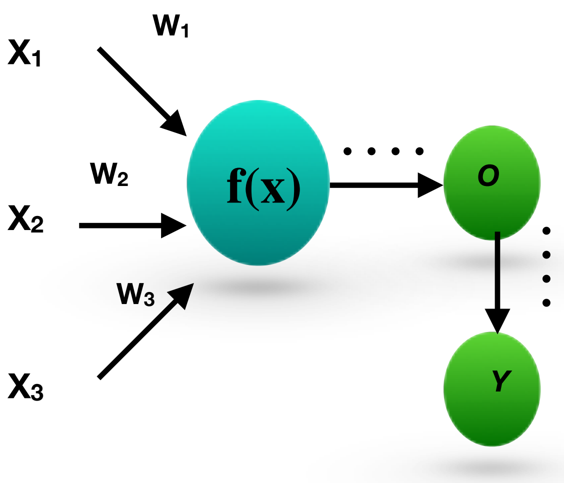
✦ **Output Layer**



Usually , the function *f(x)* used in the output layer is sigmoid function.

2.1.2 How do Artificial Neural Network Work ?

In order to understand the working of the neural network , let us consider an ANN which takes three inputs and the hidden layer of the network has only one neuron.



X_i -> input

W_i -> weights of each input

f(x) -> Activation function

O-> output of the network

Y-> Actual output

⋮ -> back propagation

After each output from the network, we will compare the O and Y and calculate ***c= 1/2(O-Y)²*** .c is a function which calculates the difference between the actual output and output given by the network. For minimising it, we will back propagate and change all the weights accordingly.

When we work with multiple sets of the inputs , we will calculate O and Y for each set and for minimising c , we will adjust the weights. This process of adjusting c and weights is known as ***Backpropagation.***

2.2 Convolution Neural Network (CNN)

A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics.
The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area.

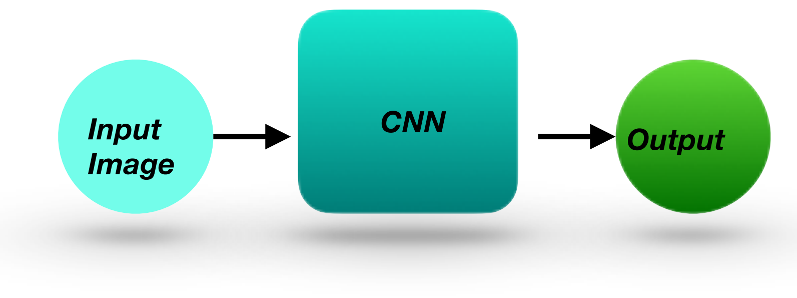
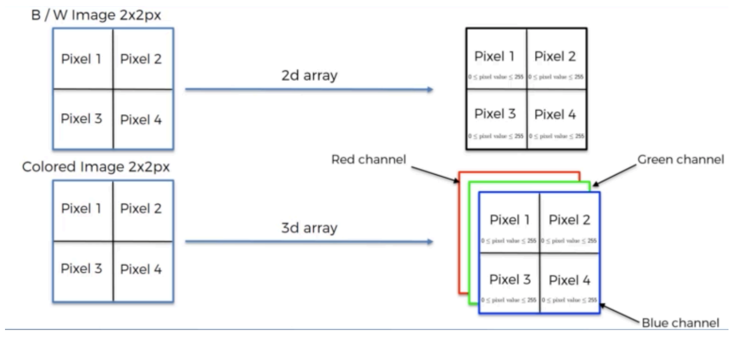


Fig:Simplified CNN



Black/White images in the CNN are converted into 2D array where as colour images are converted in 3D array of red, green and blue layers.
The various phases of the Convolutional Neural network are listed below.

- Convolution
- Max Pooling
- Flattening
- Full Connection

2.2.1 Convolution

In this phase of convolution, we apply features detector or filters to the input image and prepare the feature maps from them. The input image is provided as a matrix on which, we apply a matrix of feature known as the filter. After applying filters, we prepare a matrix containing details about that particular feature in the image known as the feature map. The feature maps are smaller in size than the input image.

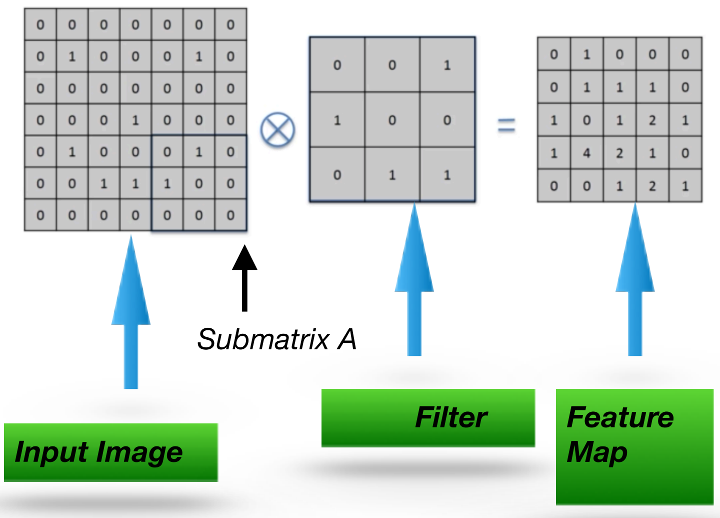
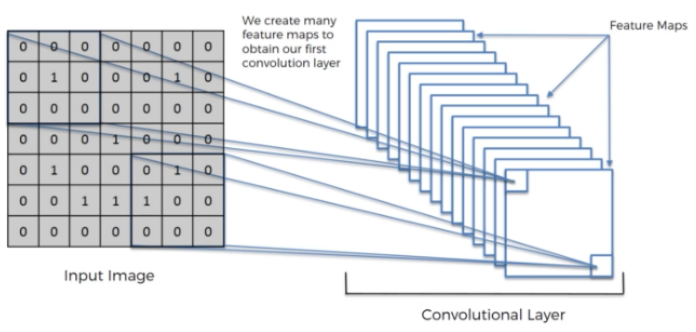


Fig: Convolution

$$\otimes \longrightarrow \sum A(i\,,j) \times Filter(i\,,j)$$

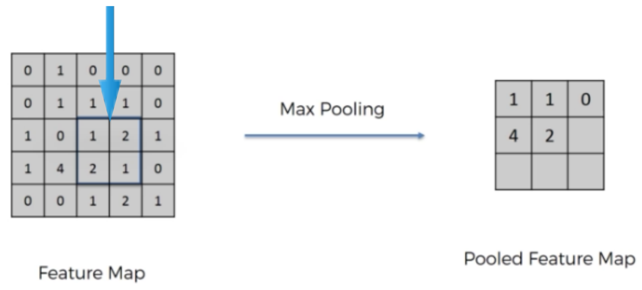
Similarly, we apply different filters to same image and prepare various features map.

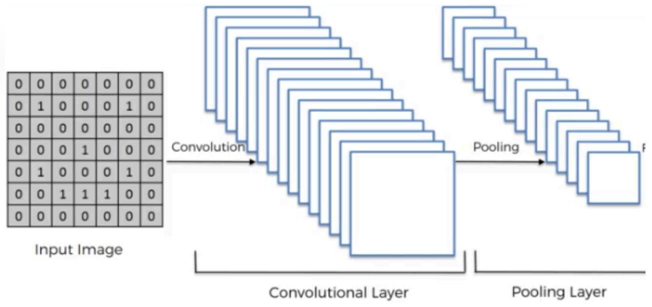

We create many feature maps to obtain our first convolution layer
Feature Maps
Input Image
Convolutional Layer

2.2.2 Max Pooling

We use max pooling so that our model can identify the features with flexibility , which means not searching for a particular feature at a specific position in all the images.It also allows us to discard unnecessary details and keeping the important ones at the same time.

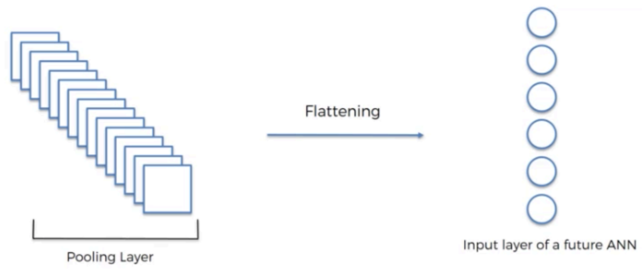
We are taking the maximum of the four values





2.2.3 Flattening

In this process, we convert the pooled layer into a 1D array.



After flattening , we will use this 1D array as an input layer to the ANN which we will add further.

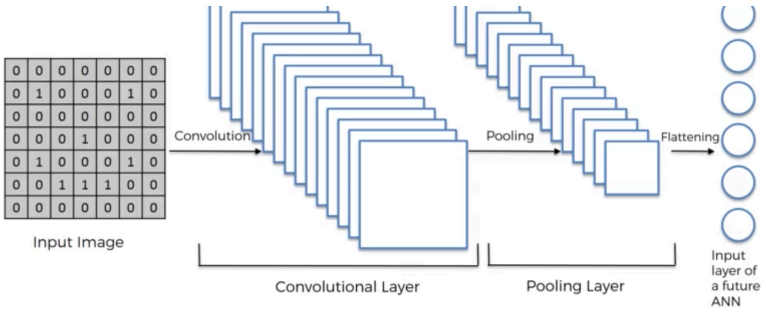
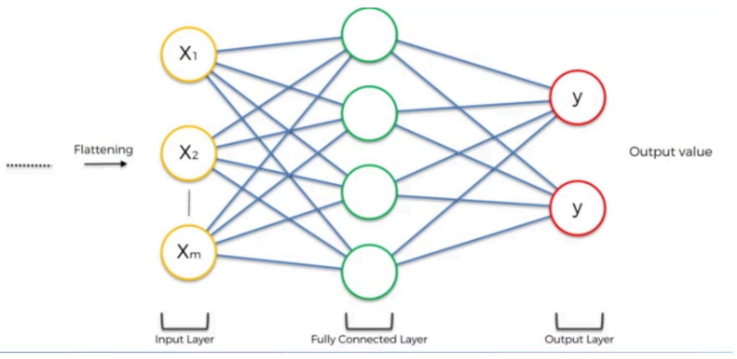


Fig: CNN upto Flattening

2.2.4 Full Connection

In this step , we will add a full Artificial Neural Network (ANN) to the Convolutional Neural Network (CNN) and train the ANN on the inputs which are provided by the array after the flattening.In ANN , hidden layer doesn't need to be fully connected, while in CNN , we use a fully connected ANN. A fully connected ANN is a neural network in which , each neuron from the different layers are connected with each other.



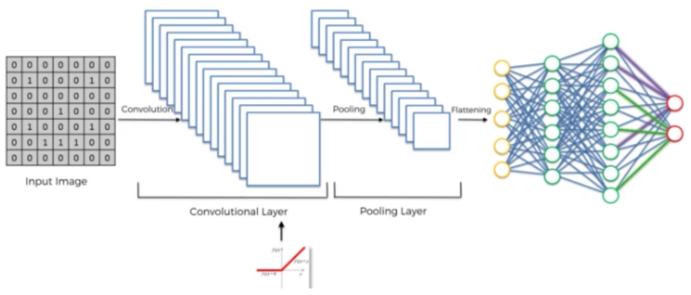


Fig:Complete Conventional Neural Network (CNN)

2.2.5 Softmax and Cross Entropy

Softmax is a mathematical function that converts a vector of numbers into a vector of probabilities, where the probabilities of each value are proportional to the relative scale of each value in the vector.

The most common use of the softmax function in applied machine learning is in its use as an activation function in a neural network model. Specifically, the network is configured to output N values, one for each class in the classification task, and the softmax function is used to normalize the outputs, converting them from weighted sum values into probabilities that sum to one. Each value in the output of the softmax function is interpreted as the probability of membership for each class.

Mathematically, we can represent Softmax function as

$$f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

Cross-entropy is a measure of the difference between two probability distributions for a given random variable or set of events.

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

$$H(p,q) = -\sum_x p(x)\,\log q(x)$$

Cross Entropy Function

2.3 Long-Short Term Memory (LSTM)

When we work with neural networks like CNN or RNN , we face two kinds of problems with the backpropagation. These problems are Exploding gradient and Vanishing gradient. In order to overcome vanishing gradient, we use ReLU, LSTM, GRUs etc. In this section, we will study the structure of LSTM and it's working.

2.3.1 Structure of LSTM

LSTM is a special kind of Recurrent Neural Network (RNN) , which is capable of learning long -term dependencies. The structure of LSTM consists a combination of sigmoid and hyperbolic tangent functions , which works as different gates and helps to memorise the long-term dependencies .

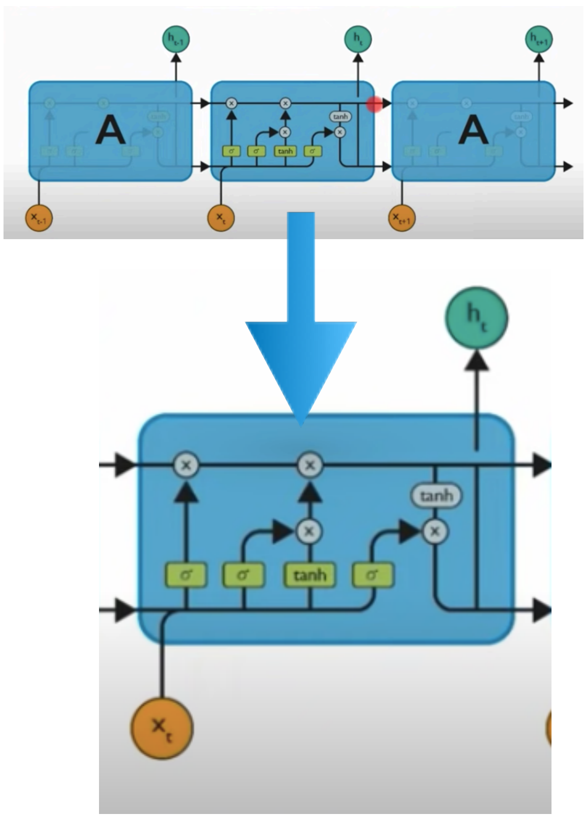
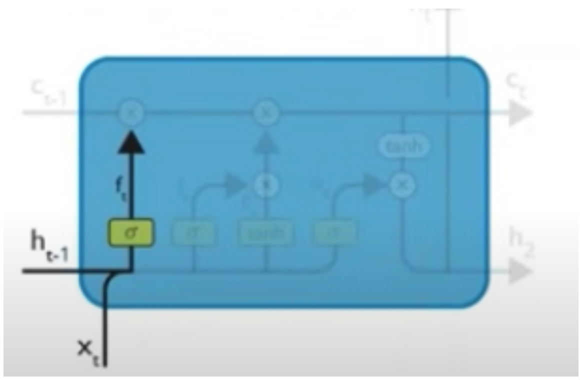


fig: LSTM

2.3.2 Working

Different steps involved in the working of LSTM are discussed below.

Step 1: The first step in the LSTM is to identify those information that are not required and will be thrown. This decision is made by a sigmoid layer called as ***Forget Gate*** layer.

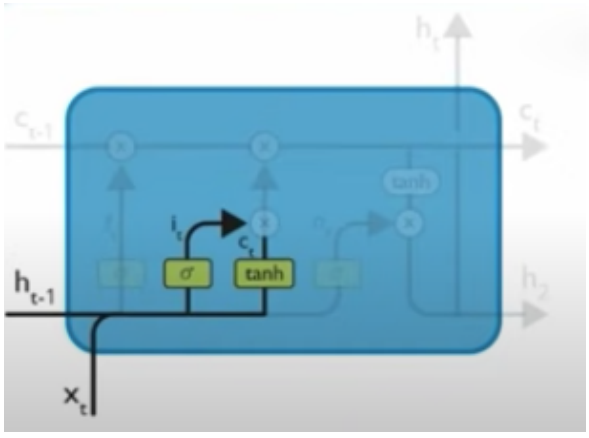


$$f_t = \sigma(w_f[h_{t-1}, x_t] + b_f)$$

W_f = weight
h_{t-1}= output from the previous time stamp
X_t = New Input
b_f = Bias

Step 2: The next step is to decide, what new information we are going to store in the cell state. This whole process comprises of the following step.

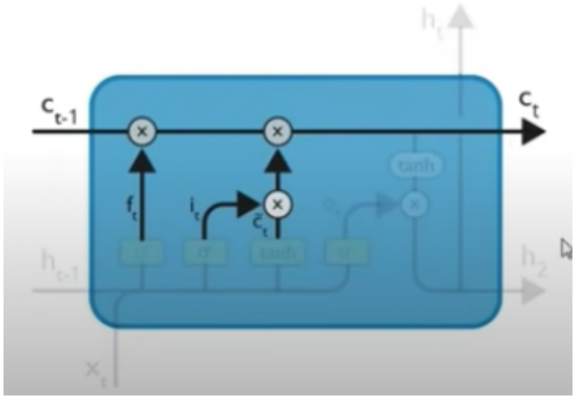
- A sigmoid layer called the "input gate layer" decides which values will be updated.
- A tanh layer creates a vector of new candidates values, that could be added to the state.



$$i_t = \sigma(w_i[h_{t-1}, x_t] + b_i)$$

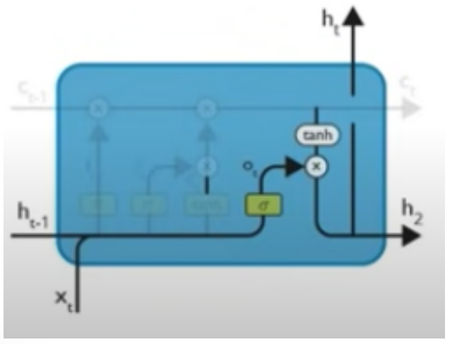
$$c\tilde{}_t = tanh(w_c[h_{t-1}, x_t] + b_c)$$

Step 3: Now , we will update the old cell state, c_{t-1} , into the new cell state ct. First, we multiply the old state c_{t-1} by f_t , forgetting the things we decided to forget earlier. Then , we add i_t*c_t. This is the new candidate values , scaled by how much we decided to update the state value.



$$c_t = f_t * c_{t-1} + i_t{}^* c\tilde{}_t$$

Step 4: We will run a sigmoid layer which decides the cell state we are going to output. Then, we put the cell state through tank and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.



$$o_t = \sigma(w_o[h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(c_t)$$

3.Long-Term Recurrent Convolutional Network (LRCN)

Long-Term Recurrent Convolutional Network (LRCN) is combination of a deep hierarchical visual feature extracted (such as CNN) with a model that can learn to recognise sequential data (inputs or outputs) , visual, linguistic , or otherwise. LRCN works by passing each visual input through a feature transformation with parameters , usually a CNN , to produce a fixed-length representation. The outputs are then passed into a recurrent sequence learning module.

In this paper, we will study and implement two vision problems which are :

1. Sequential input and output includes the tasks in which both inputs and outputs are time varying.
2. Sequential input , static output includes the task in which the input varies with time but the output is single label like running , jumping etc.

3.1 Structure of Long-Term Recurrent Convolutional Network (LRCN)

In this paper , we will look into a LRCN which is combination of CNN and LSTM. It consists of a CNN which is followed by LSTM. First , inputs are passed through the CNN which performs the recognition part. After , recognition of each frame, we will pass the outputs of the CNN to the LSTM , which then recognise the action depending on the previous as well as the current input.

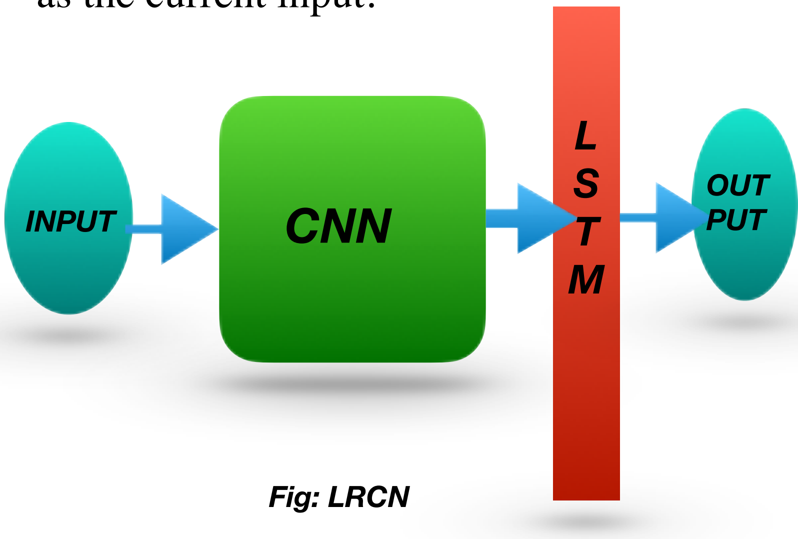


Fig: LRCN

3.2 Constructing the LRCN model in Python

The construction of LRCN model includes the various steps which are as follows.

- **Initialising the model**

We are initialising the model by using the ***Keras sequential subclass*** from the ***tensorflow*** library .

```
model = Sequential()
```

- **Adding CNN layer**

We are using the ***Keras time distributed class*** so that we can apply a CNN layer to every temporal slice of an input.

```
 model.add(TimeDistributed(Conv2D(16, (3, 3),
 padding='same',activation = 'relu'),
input_shape = (SEQUENCE_LENGTH, IMAGE_HEIGHT, IMAGE_WIDTH, 3)))

 model.add(TimeDistributed(MaxPooling2D((4, 4))))
 model.add(TimeDistributed(Dropout(0.25)))
```

Now after max pooling and dropout layer , we will flatten it.

```
model.add(TimeDistributed(Flatten()))
```

- **Adding LSTM**

We are now adding LSTM layer by using *LSTM* class of the *tensorflow library*.

```
model.add(LSTM(32))
```

- **Adding the output layer**

Activation function that we use in the output layer is the *softmax* function.

```
model.add(Dense(len(CLASSES_LIST),
activation = 'softmax'))
```