Business Club

Types of Neural Networks

Business Club Analytics Team

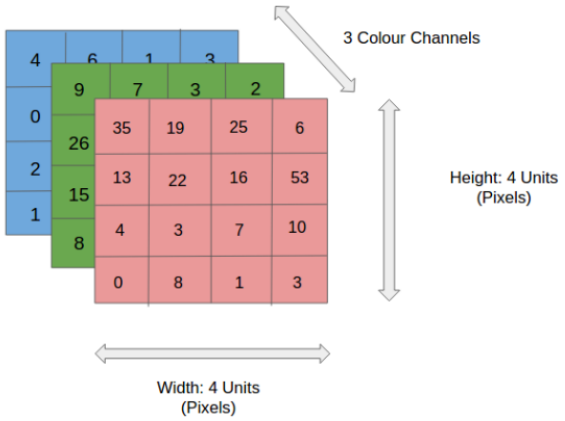
Table of Contents

Convolutional Neural Networks

Vanilla neural networks work well with tabular data. But many applications of neural networks involve image data as well, the most common task being image classification. For such purposes, Convolutional Neural Networks (CNNs) are used which are well suited to work with images. Prior to CNNs, manual time-consuming feature extraction methods were used to identify objects in images. With enough data, CNNs are able to learn such methods on their own and this applicability has led to CNNs being used widely for tasks ranging from face recognition to product recommendation.

Digital Image Representation

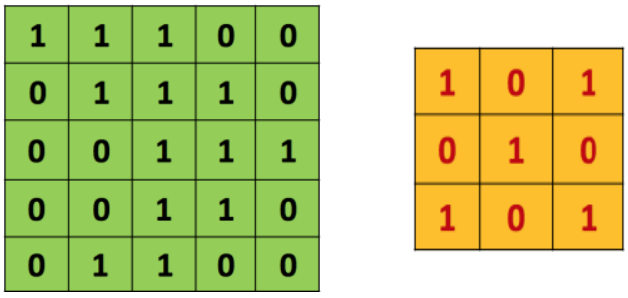
Every image can be represented as a matrix of pixel values, ranging from 0 to 255. Channel is a conventional term used to refer to the colors of an image. An image from a standard digital camera will have three channels (red, green, and blue), you can imagine those as three 2D matrices stacked over each other. An image can also have a single color channel and is usually called a grayscale image, giving only a 2D array.



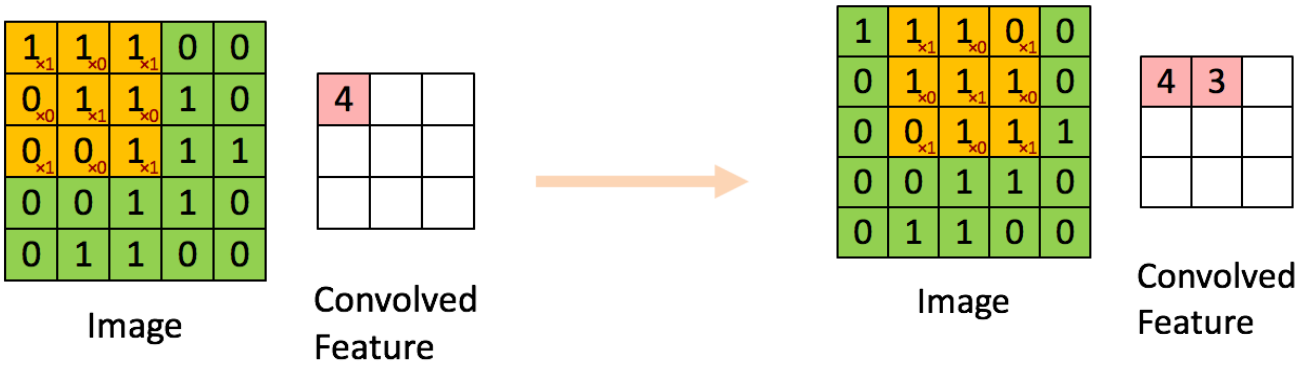
Convolutional neural networks have two kinds of layers in them: convolutional layer and pooling layer. Let’s look at the functioning of each of these operations in detail.

Convolution Operation:

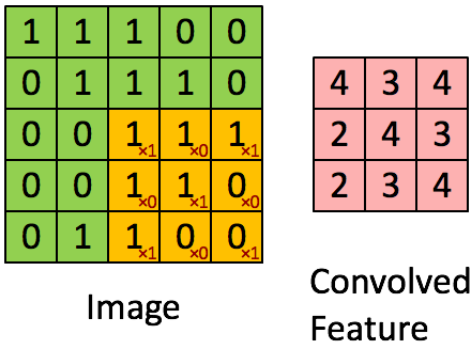
Convolutional neural networks derive their name from the convolutional operator which is an integral part of this architecture. With the help of an example, let's understand the exact working of a convolution. As discussed, each image can be represented as a matrix of pixel values. Let us consider the special case of a 5x5 grayscale image where the pixel values are either 0 or 1. We consider another matrix of the dimension 3x3 as shown. This is called a filter. We will compute the convolution of the 5x5 image with the 3x3 filter.



We slide this filter over the given image and multiply the corresponding elements. These values are summed up and stored in another matrix as shown. This filter is then shifted by 1 pixel and the same process is repeated:



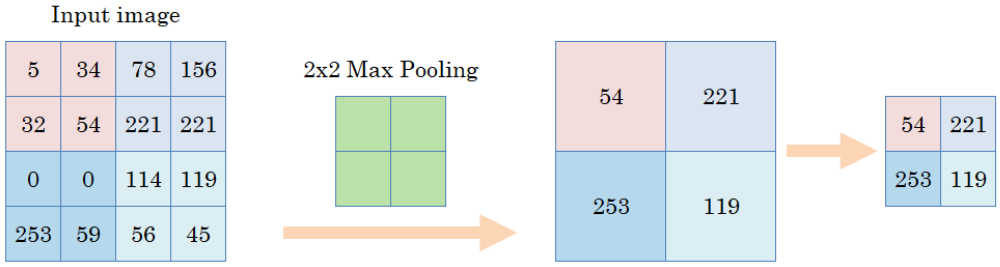
We continue doing this until every pixel has been included at least once. Note that the input image was of size 5x5 and the filter was of size 3x3, giving an output of size 3x3. We finally achieve the output shown:



This 3x3 matrix is called a kernel or a filter or sometimes a feature extractor. This is what primarily extracts important information from the image. For a lot of time, these filters were hand-engineered but with the advent of convolutional neural networks, the idea that these feature extractors could be learned by the machine itself became prominent.

Max Pooling Operation:

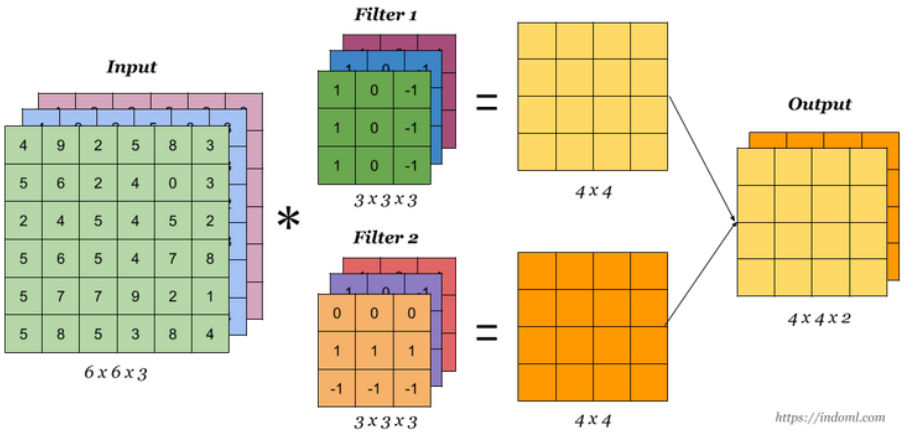
Here we take groups of pixels (for example, groups of 2x2 pixels) and perform an aggregation over them. This aggregation in the case of max-pooling is just the maximum value among that group. An example of the same can be seen below.



But how is this helpful? This downsamples the image to a smaller size (thereby reducing the parameters required) and also intensifies the higher valued pixels. This helps to reduce the noise from the output. It also helps to achieve translation invariance which is very important for tasks like image classification where the object can be anywhere in the image.

Basic Architecture of a CNN:

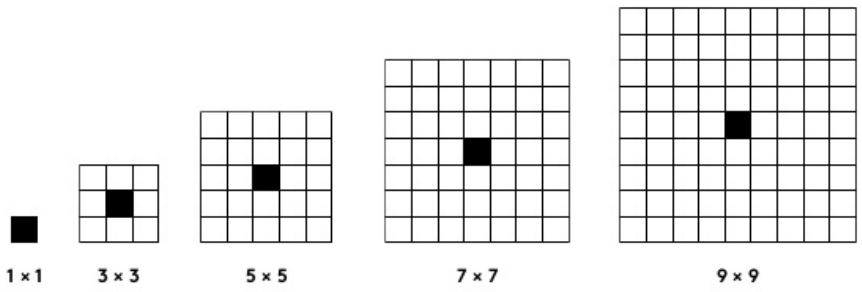
Earlier we had talked about convolutions on a grayscale image, let's see how that applies to an RGB image. Now we have a 3D input so to process that we take a 3D filter as well with the same principle of application as before. We can also have a number of filters be applied and then later stack up the outputs.



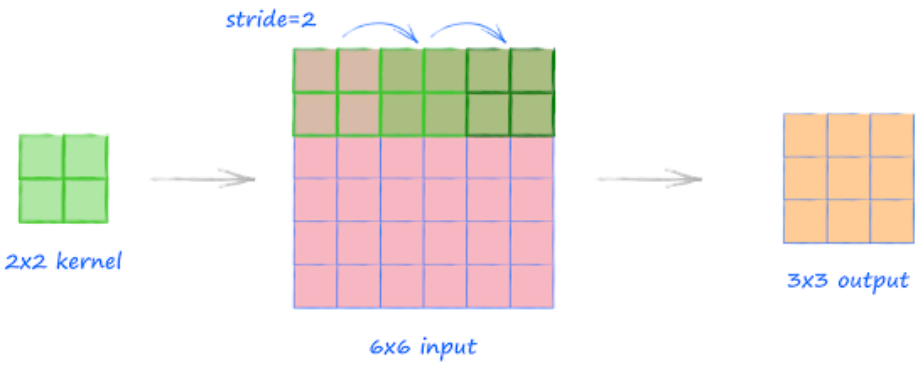
The basic principle of CNN is that the filter values are made variables that are to be learned. As these are primarily the feature extractors, CNNs are given the freedom to learn whatever types of features they deem fit for the task at hand. Once the convolution operation occurs, we add biases to each of the 2D outputs and apply an activation function, which gives the activation map. All these constitute a single layer of a convolutional neural network.

Let's look at the parameters related to this convolutional operation:

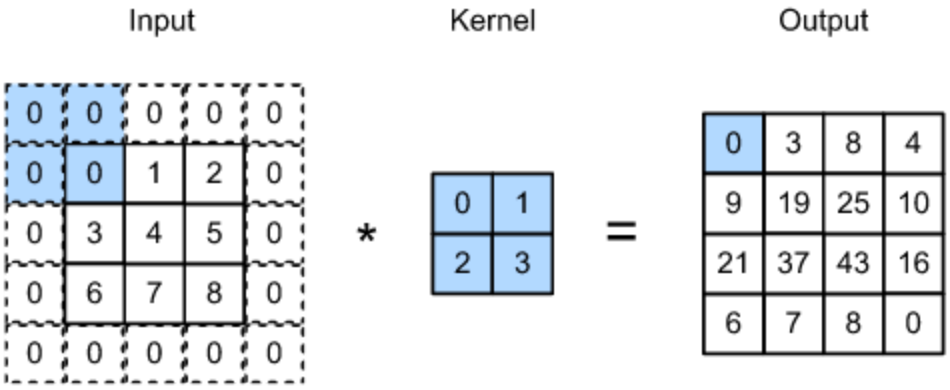
- **Filter size**: f, this can range from as small as 1×1 to even as large as 11×11 and more.



- **Stride:** s, i.e., how many pixels should the filter move in each step while sliding across the input volume. An example of using stride 2 with a 2x2 kernel is shown below



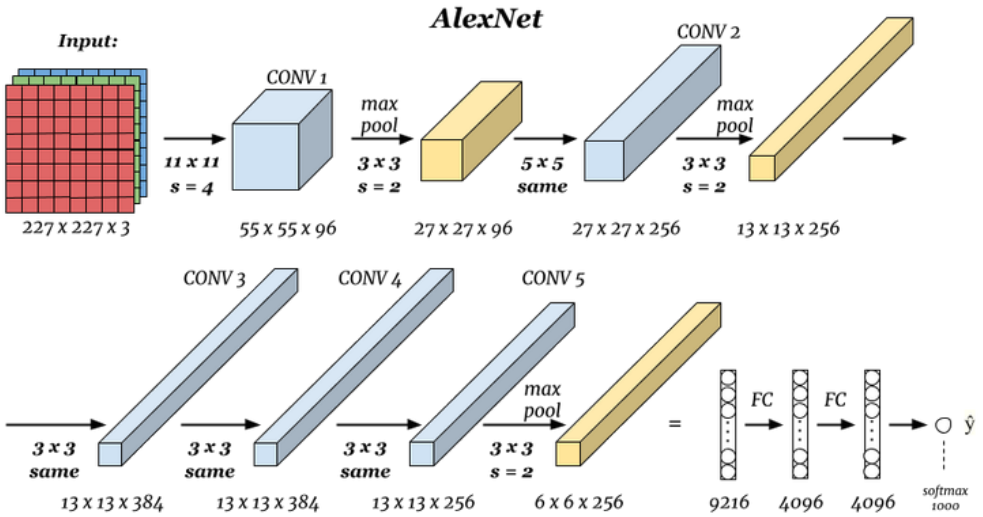
- **Padding:** p, adding layers of zeros to the input volume so as to get the same sized output (such a case is also called *same*) or at least a bigger output than normal.



Let's say the input volume is of size $H \times W \times C$, then the size of the output volume can be related by the simple formulae:

$$H' = \left[\frac{H + 2p - f}{s} + 1\right] \quad , \quad W' = \left[\frac{W + 2p - f}{s} + 1\right]$$

Now as we've become acquainted with the operations involved in a CNN, let's delve deep into one of the most famous CNN architecture: AlexNet



AlexNet used a 227×227 RGB image for classification into one of 1000 labels. It started off with using large filters (11×11) so as to downsample more, later using max-pooling too. You might have noticed a pattern of convolution operation then pooling. This works well due to several reasons, the most prominent being: noise reduction and the requirement of fewer parameters.

In each of its steps, it used the ReLU activation function. After a series of convolutional and pooling operations, it was reduced to a size of $6 \times 6 \times 256$ which was flattened into a single vector of 9216 elements. This is treated as the input layer for artificial neural networks and is connected to 2 hidden layers in succession. Ultimately, the softmax function is utilized to classify the vector into one of the 1000 labels.

It had over 60 million parameters and took about a week to fully train on the ImageNet dataset.

Why are CNNs better than ANNs?

Convolutions are very useful for two primary reasons:

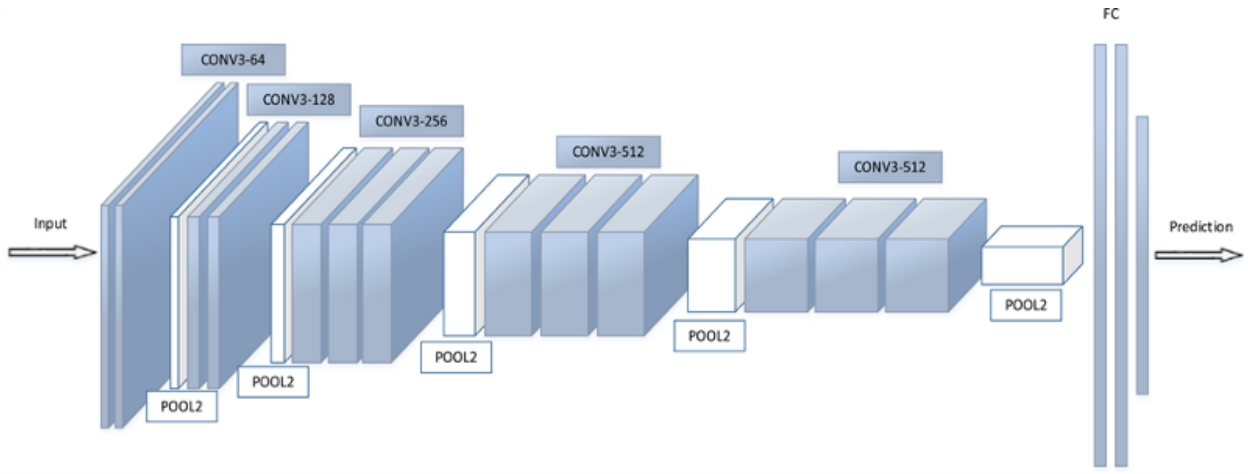
- **Parameter sharing:** A filter that's useful in one part of the image is most probably useful in another part of the image. For example, an edge detector filter will be useful regardless of the space it operates on. This highly reduces the number of parameters used compared to a similar ANN.

  Let's say we have $32 \times 32 \times 3$ input image and convolve it with six 5×5 filters. That would result in an output of dimensions $28 \times 28 \times 6$. Here, the convolution operation involves 152 parameters (each filter contributing 25 filter values and 1 bias value). Comparing to an ANN, a vanilla neural network would require more than 14 million parameters to achieve the same.

- **The sparsity of Connections:** In each layer, the output value in each point of the output volume only depends on a small number of inputs. In other words, the output values are quite independent of each other and thereby inhibit overfitting during network training. In contrast, in an ANN, each output value is a result of the combination of all the corresponding input values.

Case Study of a CNN Architecture: VGG-16

VGG-16 was proposed by the Visual Geometry Group Lab of Oxford University in 2014 and was considered state-of-the-art at that time for the task of image classification. It completely dominated the 2014 ImageNet Large Scale Visual Recognition Challenge (ILSVRC), a popular image classification challenge.



Convolutional neural networks have a lot of hyperparameters, what filter size to choose, stride of the filter, padding, how many layers, etc. VGG-16 proposed that instead of optimizing so many hyperparameters, it will predefine the pooling and convolutional layers to be used when required: 3×3 convolution (stride 1, padding 1) and 2×2 max-pooling (stride 2), which were referred to as CONV3 and POOL2 respectively.

The basic idea of VGG was that stacking multiple such small convolutional layers will produce the same receptive field as using a large filter once with more non-linearity, but with fewer parameters.

Let's compare a single 7×7 convolution with a stack of three 3×3 convolutions, where the number of filters to use is C:

No. of weights for 7×7 convolution = $49C^2$

No. of weights for three 3×3 convolutions = $27C^2$

Thus, we'll achieve more non-linearity along with fewer parameters by this method.

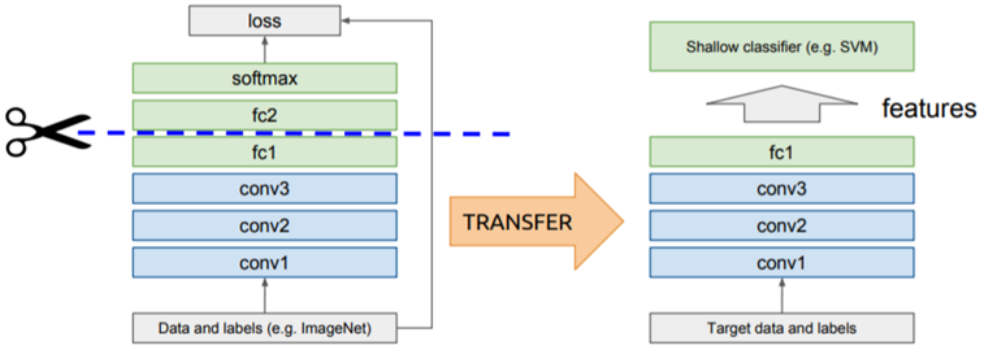
This model was trained on about 14 million images from the ImageNet dataset for classification into 1000 labels. They employed 13 convolution/pooling layers and 3 fully connected layers at the end as the classification head, giving the name VGG-16. It has about 135 million parameters to be optimized and at the time it was proposed, took about 2-3 weeks to fully train. This was considered a very deep network during that time, though now networks with even 50 layers are common. It illustrated the fact that representation depth is beneficial for classification accuracy and state-of-the-art performance can be achieved with substantially increased depth.

As this model was trained on such a wide variety of images, it learned to extract general features from the images too. This leads to an important concept of deep learning called **Transfer Learning** which helps one to construct an accurate model using prior knowledge even when we have fewer data.

Transfer Learning

It is a popular approach in deep learning where pre-trained models are used as the starting point on various tasks given the vast compute and time resources required to develop neural network models on these problems and from the huge jumps in accuracy that they provide on related problems. The applicability of this method can be with an example using VGG-16.

Say we have the VGG-16 model pre-trained on the ImageNet dataset. It has seen millions of images of 1000 categories and has learned efficient feature extractors to classify them. These feature extractors might be generally applicable to similar tasks too. So we retain the convolutional layers of this model and replace the last fully connected layer with a shallow classifier.



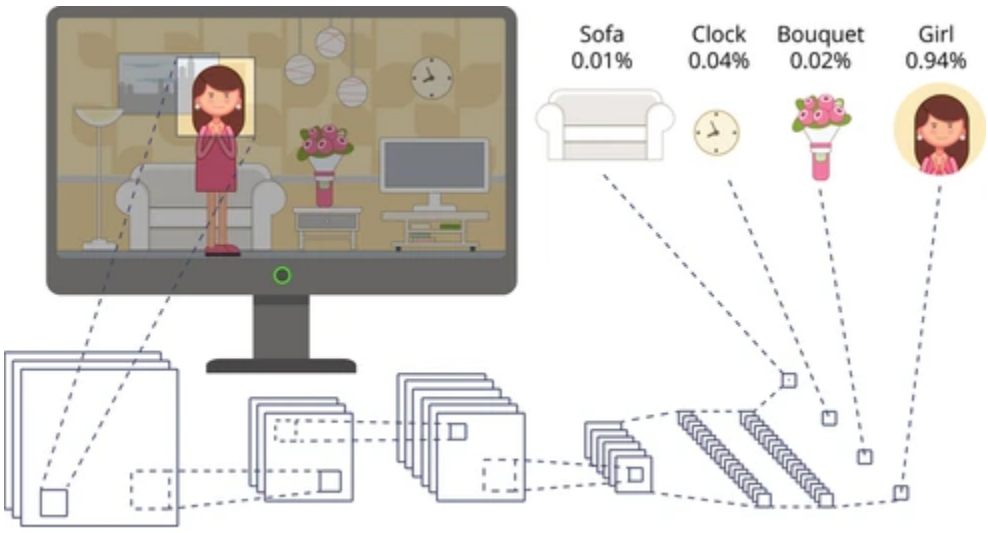
Imagine you have a dataset of bicycles and unicycles and you want a model to differentiate between the two. However, the dataset is only a few thousand images. In such a case, you'll take the VGG-16 model pre-trained on the ImageNet dataset, replace the classification head with a shallow classifier and train only that part using your small dataset.

This works very well in practice and consistently outperforms very specialized task-focused deep learning models.

Application of CNNs in Industry:

CNNs are widely used in the industry for applications ranging from face recognition to product recommendations to improving video calls. It has found its way into various domains and is being applied in various fields.

Computer Vision Task	Real Life Application
Image Classification	- Image Tagging enables easier searching of images and is used by Facebook - Visual Search involves matching an input image with the available database and is employed by Google for reverse image search - Categorising listing photos of houses and rooms at Airbnb for an easier exploration of the facility
Recommendation Engine	- The image feature vectors are used to find visually similar images so as to recommend products to the customers
Face Recognition	- It serves as a streamlining of the process of tagging people in photos of various social media websites - Snapchat and Instagram filters heavily rely on this to give funky looking images
Optical Character Recognition	- Many document scanner application come with the facility of OCR and help convert the image text form to normal text - Also employed for number plate detection and data entry in business documents



To show off how some of this is actually implemented, we have prepared a demo in the form of notebooks on Google Colaboratory which can be accessed from our GitHub repository: link

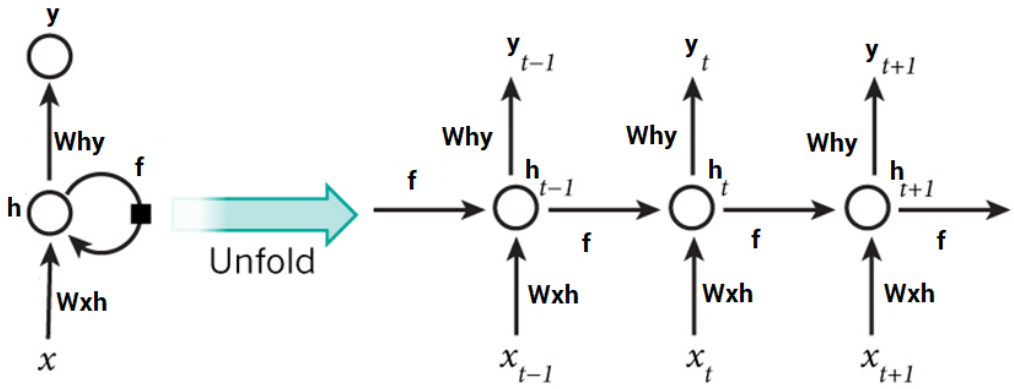
Recurrent Neural Networks

The intuition behind introducing Neural Networks was to replicate the working of the human brain. Feedforward vanilla neural networks did not do justice to the idea as the model does not remember the inputs given in the previous iterations, which could affect the detection of certain trends. They worked under the assumption that the inputs after every iteration were independent of each other. Which is not necessarily true in real-life applications. For example, while predicting the stock price, remembering the previous data is very important as previous stock prices play a crucial role in predicting future prices.
Thus, Recurrent neural networks were introduced. They are architectured in a way that allows them to retain the information gained from the previous computations.

Basic Architecture of an RNN:

As we know, RNN models take the previous information into consideration while calculating the output for the current state. Thus the network is formed by connecting such repeated cells.

Given below is the unfolded version of RNN-



Input state - x_t is the current input state, it is the sequential data that the network receives in every iteration.

Hidden state - h_t is the current hidden state of the recurrent neuron. It is essentially the memory of the network which is passed to the next cell as one of the inputs.

Output state - y_t is the current output state.

So the current state is a function of the previous hidden state and current input state that can be written as,

$$x_t = f(x_t, h_{t-1})$$

Assuming the activation function to be tanh and taking
$\boldsymbol{W_{xh}}$ - Weight at input neuron
$\boldsymbol{W_{hh}}$ - Weight at the recurrent neuron

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

Note - The above network just takes into account the immediately previous state. We can also take multiple multiple states according to the application of the model.

The output state is obtained by taking into consideration just the current recurrent state.

$$y_t = f(h_t)$$

Assuming the function to be sigmoid to obtain the probabilities over output and taking
$\boldsymbol{W_{hy}}$ - Weight at output neuron

$$y_t = sigmoid(W_{hy} * h_t)$$

Vanishing and exploding gradients problems:

The problem of vanishing and exploding gradients in RNN's is caused essentially due to repeated multiplication of weight while computing derivatives for backpropagation.

While updating the weights for reducing loss function, we have to take into consideration h_t ,which is obtained by multiplying W_{hh} and h_{t-1}. Now, h_{t-1} is further obtained by multiplying W_{hh} and h_{t-2} and this process goes on.

If W_{hh} is too small, then the gradients will decrease exponentially, making it harder for the model to update the weights as we go deeper in the network. This problem is called the vanishing gradient problem.

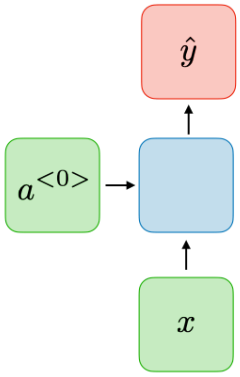
If W_{hh} is too large, then the gradients will increase exponentially, which results in drastic updates in the weights, making the model unstable. This problem is called the exploding gradient problem.

Vanishing Gradient	Exploding Gradient
1. Randomly initialising the weights 2. Using specialized RNN units like LSTMs/GRUs with gates to discard irrelevant past information 3. Using Echo State networks which is a type of reservoir computer with sparsely connected hidden layer	1. Using Gradient Clipping to prevent any gradient to have a norm greater than the threshold 2. Adding L1 or L2 regularisations to the loss function to penalize the model 3. Early Stopping to avoid overshooting of gradients

Given here are some ways to avoid the problem

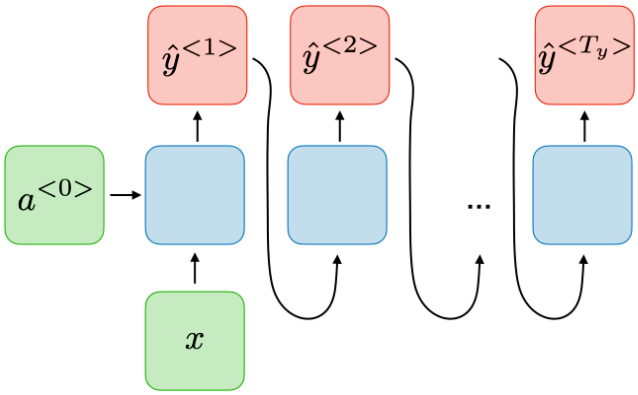
Types of RNN:

- **One to One:** They are the most basic architecture. This network has just a single output for a single input.



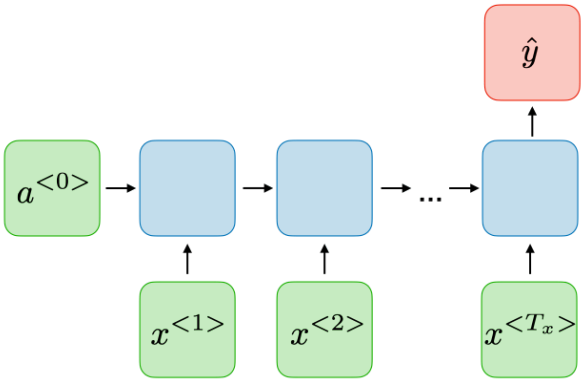
Such models are used in traditional neural network models.

- **One to Many:** This architecture is used when we produce multiple outputs from a single input.



Such models are used in image captioning where we take an image of a given size as input and give a sentence of variable length as output.

- **Many to One:** This architecture is used when we incorporate multiple inputs to produce a single output.

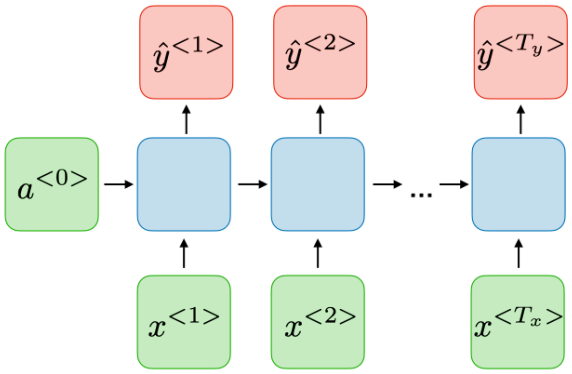


  Such models are generally used in sentiment analysis wherein we give a paragraph or sentence as input and classify them as 0 or 1 depending upon the negative or positive sentiment.

- **Many to Many:** This architecture is used when we produce multiple outputs using multiple inputs.

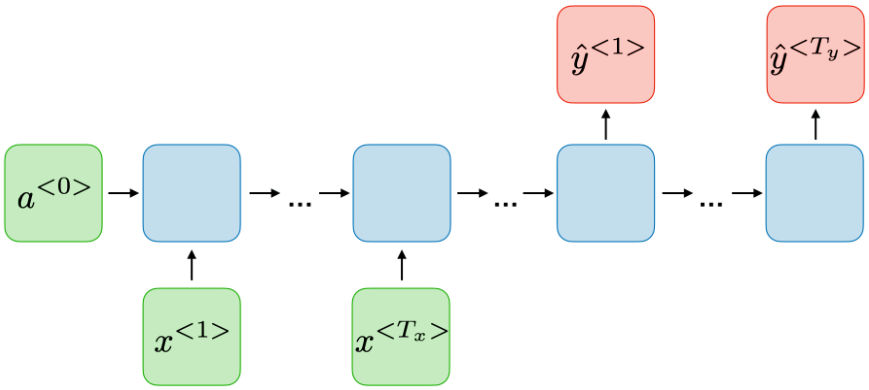
  Depending upon the number of inputs and outputs, they can be further classified into 2 categories, which we have discussed in detail below:

1. **Many to Many(layers of input and outputs are same)**



Such models are used in Named-Entity Recognition.

2. **Many to Many(layers of input and outputs are different)**



Such models are used in machine translation where both the input and output length is variable.

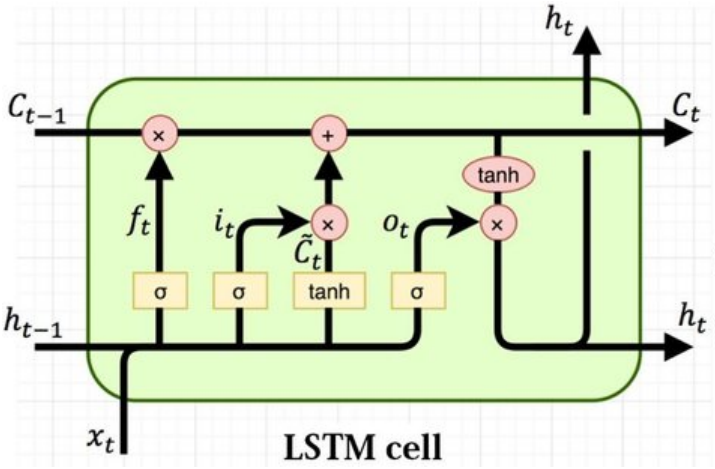
Specialized units of RNN:

Long Short-term memory(LSTM)

LSTMs are special types of RNNs that were designed to overcome the problem of short-term dependencies of vanilla RNN models. They have specialized gates that allow the models to retain the information over a longer period and overcome the problem of vanishing gradients as they don't multiply the weights directly while minimizing the loss function.

Architecture:

The LSTM block is made of 3 gates, each performing a specific task.



- **Forget gate**: This gate removes the unnecessary chunk of information that may no longer be required by the LSTM cell.
  It multiplies the output by 0 if the information needs to be "forgotten" and 1 if the information needs to be "remembered".

- **Input gate:** This gate is responsible for passing the information that it gathers in the current timestamp from the data entered in the cell.

- **Output gate:** This gate is responsible for calculating the value of the next hidden state.

Let,
$\mathbf{x_t}$ be the input vector at the current timestamp
$\mathbf{f_t}$ be the forget gate
$\mathbf{i_t}$ be the input gate
$\mathbf{o_t}$ be the output gate
$\mathbf{h_t}$ be the hidden state vector at the current timestamp
$\mathbf{C_t}$ be the cell state vector at the current timestamp
$\mathbf{C'_t}$ be the cell input activation vector at the current timestamp
$\mathbf{W_x}$ and $\mathbf{U_x}$ are the weights for the "x" gate.

Now all three gates are a function of current input and previously hidden vector. We pass them through the sigmoid function to obtain the probability vector between 0 to 1.

$$f_t = sigmoid(W_f * x_t + U_f * h_{t-1})$$

$$i_t = sigmoid(W_i * x_t + U_i * h_{t-1})$$

$$o_t = sigmoid(W_o * x_t + U_o * h_{t-1})$$

The cell input activation vector provides the new information taking the input and hidden state as vectors and using tanh activation function.

Gradient computing is less expensive and handles vanishing gradient problems much better.

$$C'_t = tanh(W_c * x_t + U_c * h_{t-1})$$

The updated cell state is given as -

$$C_t = f_t \circ C_{t-1} + i_t \circ C'_{t-1}$$

$$h_t = o_t \circ tanh(C_t)$$

If we want the output, we can just apply softmax to the hidden state vector.

$$O_t = softmax(h_t)$$

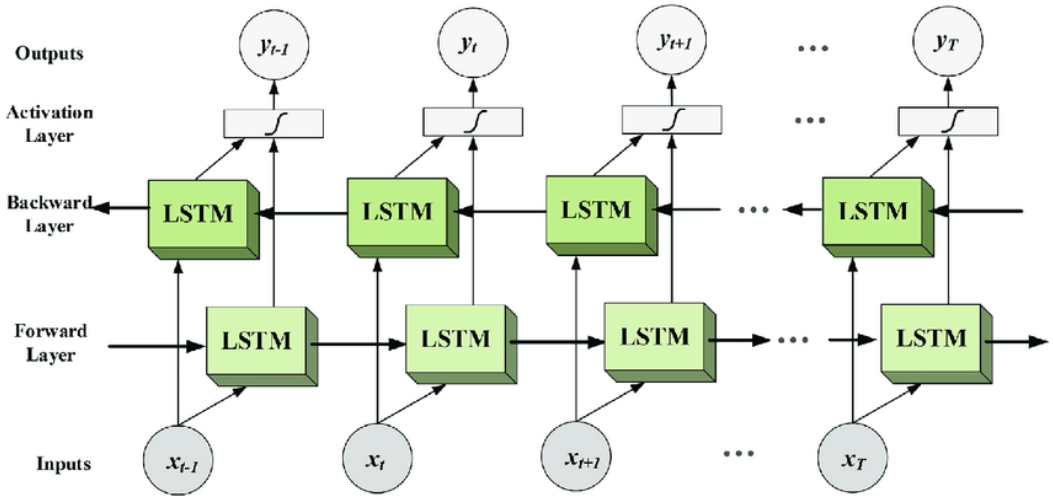
LSTMs are applicable wherever we work with sequential data like time series forecasting, video/speech/text recognition,

Depending on the type of connections, LSTMs can be classified as -

- **Unidirectional LSTM(Uni-LSTM):** These are the traditional LSTMs (explained above) with just one network to access and store the information from the past while predicting future information.
- **Bidirectional LSTM(Bi-LSTM):** These are a special type of LSTMs with two independent networks working simultaneously. One network works in the forward direction, storing the information that it gets from

past data while the other works in the reverse direction, storing the information it gets from future data. This LSTM takes into consideration both the past and future context while making predictions.

Structure of Bi-LSTM -



Although Bidirectional LSTMs are computationally more expensive since we are practically running two models simultaneously, BiLSTMs are way more efficient than the Uni-LSTMs as they understand the context much better.

For instances,

“In my opinion, Robert Downey Jr. is arguably the best fit for Iron Man.”

“In my opinion, Robert Baratheon was the most impulsive character in Game of Thrones.”

Here, just by the word ”Robert”, we can not figure out if the user is talking about the actor or the GOT character. Thus for completing the sentence after ”In my opinion, Robert _____”, we need to have the context provided in the latter part.

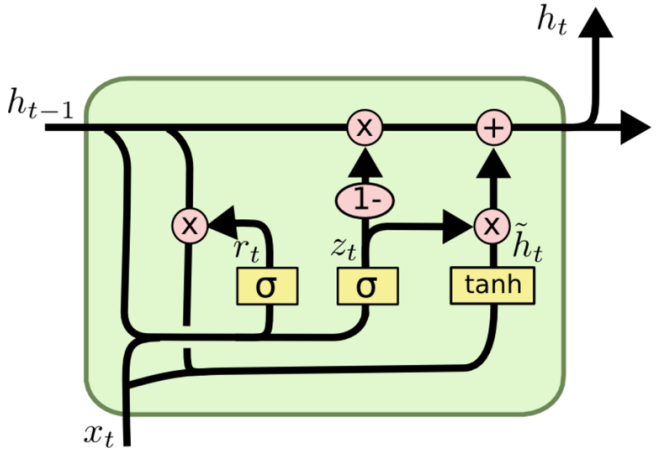
This is where Bidirectional LSTMs will play an important role as we need both the forward as well as the reverse sequential information. Hence depending on the use and computational feasibility, we can use Uni-LSTM, Bi-LSTM, or a hybrid model of them.

Gated Recurrent Unit(GRU)

GRUs are a type of RNNs that, like LSTMs, use gates to overcome the problem of vanishing gradient descent. Unlike LSTMs, they have just two gates and give significantly better results as compared to LSTMs for small datasets and are less complex hence faster in nature.

Architecture:

The GRU block is made of 2 gates, each performing a specific task.



- **Reset gate**: This gate is responsible for controlling the past information that needs to be forgotten.

- **Update gate:** This gate is responsible for deciding the information to be retained and passed to the next cell.

Let,
$\mathbf{i_t}$ be the input vector at the current timestamp
$\mathbf{r_t}$ be the reset gate
$\mathbf{z_t}$ be the update gate
$\mathbf{h_t}$ be the hidden state vector at the current timestamp
$\mathbf{h'_t}$ be the candidate hidden vector at the current timestamp
$\mathbf{W_x}$ and $\mathbf{U_x}$ are the weights for the "x" gate.

Now both the gates are a function of current input and previously hidden vector. We pass them through the sigmoid function to obtain the probability vector between 0 to 1.

$$r_t = sigmoid(W_r * x_t + U_r * h_{t-1})$$

$$z_t = sigmoid(W_z * x_t + U_z * h_{t-1})$$

Candidate hidden state is a function of input state and previous hidden state multiplied by the reset gate to control the influence from the previous hidden state.

$$h'_t = tanh(W_h * x_t + U_h * (h_{t-1} \circ r_t))$$

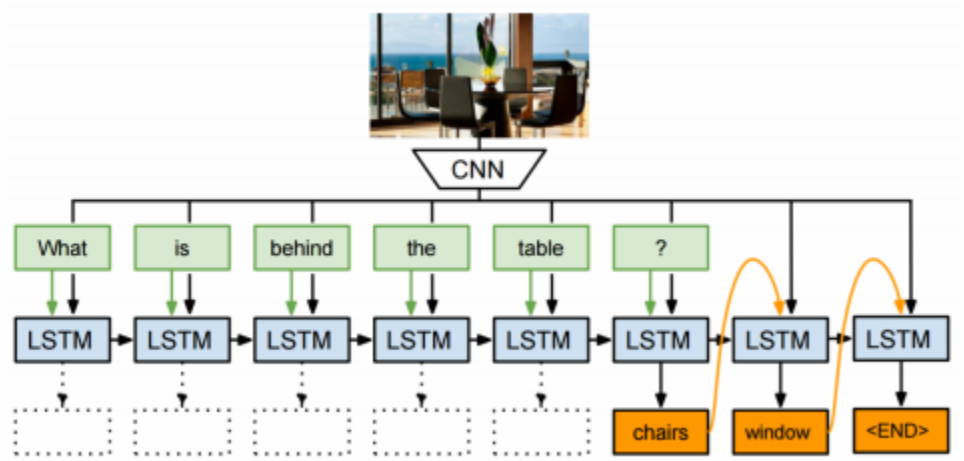
After calculating the candidate hidden state, we calculate the hidden state which is a function of the update gate, previous hidden state, and candidate hidden state.

$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ h'_t$$

Application of RNNs in Industry:

Rnns are used in various fields due to their ability to work with sequential data, ranging from Time series forecasting to Speech/ text recognition.

RNN Task	Real Life Application
Sentiment Analysis	- Helps in understanding and inferring public opinions or sentiments regarding a particular topic. - Used in analyzing customers feedback or monitoring reviews on a post on social media.
Time Series Forecasting	- Analyzing past historical data to find trends and patterns and predict the future outcome. - Used in Stock price prediction.
Speech Recognition	- Understanding and identifying words and converting them into text - Used in technologies like Siri, Alexa, Google Assistant, Cortana etc.
Machine Translation	- Converting text from one language to another. - Used in Google Translate.



To show off how LSTMs actually implemented, we have prepared a demo in the form of notebooks on Google Colaboratory which can be accessed from our GitHub repository: link