Exploration of Convolution Neural Network for Computer Vision Application

Project Report

Prepared By,

Pritam Bhakta

&

Pallabi Mondal

Guided By,

Pranab Singha Roy

Place: Variable Energy Cyclotron Centre,

Kolkata, West Bengal, 700064

Duration : 18th June – 18th July

Year : 2019

**Table of Content**

Introduction to Deep Learning…………………………………3

Machine Learning Vs Deep Learning………………………….5

How Deep Learning Works……………………………………6

What is a Neural Network……………………………………..7

## **Neural Network Architectures…………………………………..7**

## Single Layer Perceptron (SLP)

## Multi layer Perceptron (MLP)

## Artificial Neural Network (ANN)

## Recurrent Neural Network (RNN)

## Convolutional Neural Network (CNN)

### **What is Computer Vision……………………………………….11**

### How Computer Vision Works with CNN………………………12

* 1. Why ConvNets over Feed-Forward Neural Nets?
  2. Input Image
  3. Convolutional Layer – The Kernel
  4. Strides
  5. Pooling Layer
  6. Fully Connected Layer

Activation Function…………………………………………….17

* 1. Introduction
  2. What is an Activation Function?
  3. Popular types of activation functions and when to use them
     1. Binary Step Function
     2. Linear Function
     3. Sigmoid
     4. Tanh
     5. ReLU
     6. Leaky ReLU
     7. Softmax
  4. Choosing the right Activation Function

Optimizers………………………………………………………25

* **Role of an optimizer**
  + 1. **Gradient Descent**
    2. Stochastic **Gradient Descent**
    3. Mini Batch **Gradient Descent**
    4. **RMSprop**
    5. **Adagrad**
    6. **Adadelta**
    7. **Adam**
    8. **Nadam**

Loss Function…………………………………………………...29

* 1. Regression Problem
  2. Binary Classification Problem
  3. Multi-Class Classification Problem
     1. Mean Squared Error Loss
     2. Binary Cross Entropy-Loss
     3. Multi-class Cross Entropy-Loss
     4. Sparse Multi-class Cross Entropy-Loss
     5. Kullback Leibler Divergence Loss

Project………………………………………………………….33

1. Dataset
2. Project Objective
3. Dataset Description
4. Environment Used
5. Google Colaboratory

Why we use colab?

1. Keras

Why we use Keras UPI?

1. Import Packages
2. Building the Model
3. Data Augmentation

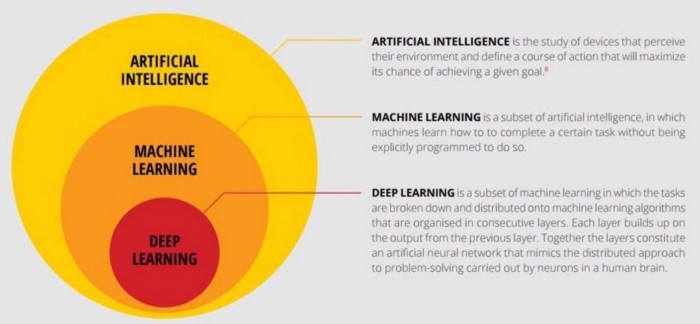
Why we do Data Augmentation

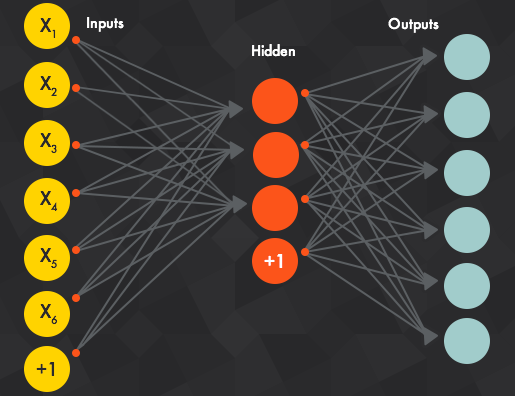
1. Model using Adam Optimizer
2. Model using SGD Optimizer
3. Feature Viduslization
4. Model Performance

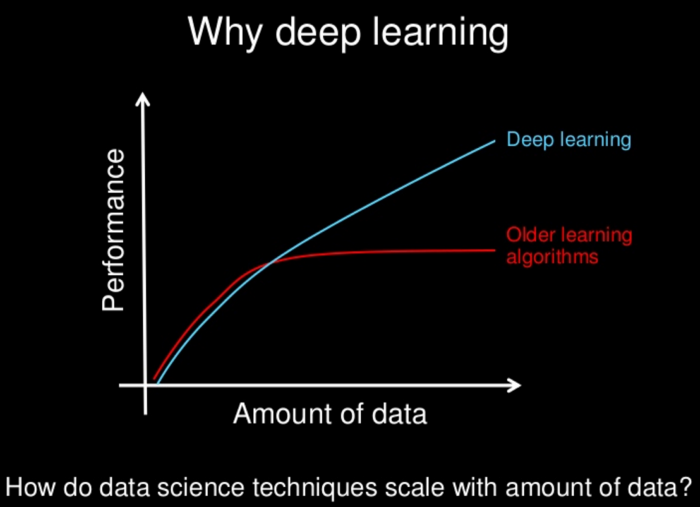
Confusion Matrix

1. Test the Model with our own Dataset

* **Introduction to Deep Learning**

Deep learning is a sub-field of machine learning dealing with algorithms inspired by the structure and function of the brain called artificial neural networks. In other words, It mirrors the functioning of our brains. Deep learning algorithms are similar to how nervous system structured where each neuron connected each other and passing information.

Deep learning models work in layers and a typical model at least have three layers. Each layer accepts the information from previous and pass it on to the next one.

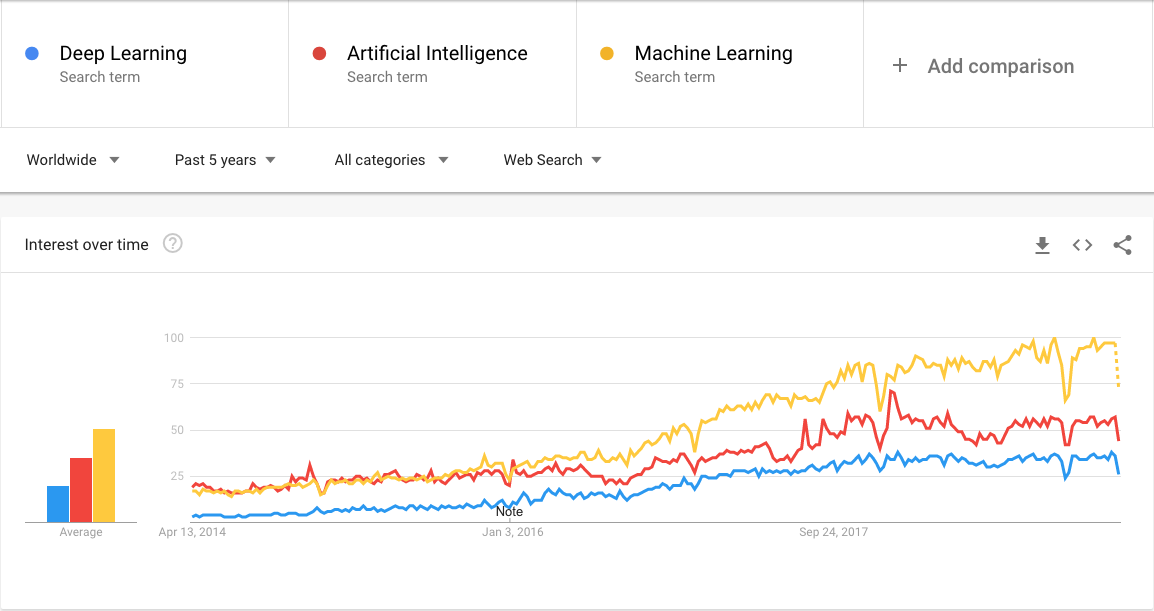


Deep learning models tend to perform well with amount ofdata wheras old machine learning models stops improving after a saturation point. One of differences between machine learning and deep learning model is on the feature extraction area. Feature extraction is done by human in machine learning whereas deep learning model figure out by itself.



* **Machine Learning Vs Deep Learning**

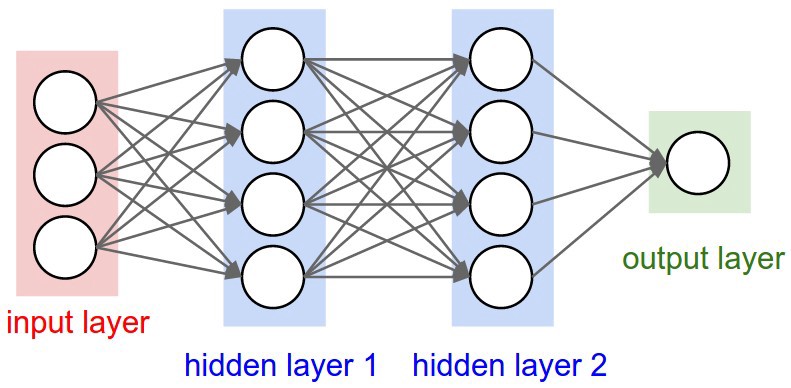
|  |  |  |
| --- | --- | --- |
|  | **Machine Learning** | **Deep Learning** |
| **Data Dependencies** | Excellent performances on a small/medium dataset | Excellent performance on a big dataset |
| **Hardware dependencies** | Work on a low-end machine. | Requires powerful machine, preferably with GPU: DL performs a significant amount of matrix multiplication |
| **Feature engineering** | Need to understand the features that represent the data | No need to understand the best feature that represents the data |
| **Execution time** | From few minutes to hours | No need to understand the best feature that represents the data. |
| **Interpretability** | Some algorithms are easy to interpret (logistic, decision tree), some are almost impossible (SVM, XGBoost) | Difficult to impossible |

****

* **How Deep Learning Works**

Most deep learning methods use **neural network** architectures, which is why deep learning models are often referred to as **deep neural networks**.

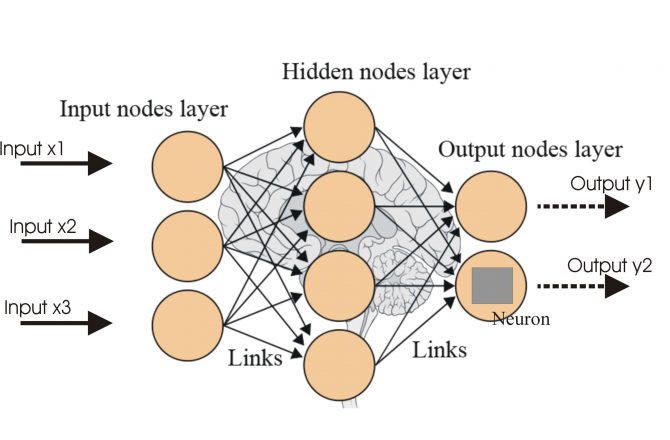
The term “deep” usually refers to the number of hidden layers in the neural network. Traditional neural networks only contain 2-3 hidden layers, while deep networks can have as many as 150.

Deep learning models are trained by using large sets of labeled data and neural network architectures that learn features directly from the data without the need for manual feature extraction.

* **What is a Neural Network?**

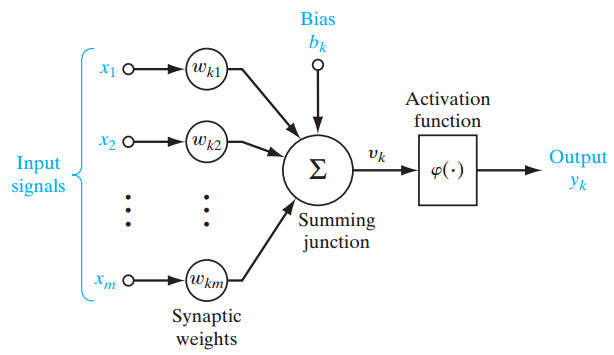
A neural network is a type of machine learning which models itself after the human brain. This creates an artificial neural network that via an algorithm allows the computer to learn by incorporating new data.

While there are plenty of artificial intelligence algorithms these days, neural networks are able to perform what has been termed deep learning. While the basic unit of the brain is the neuron, the essential building block of an artificial neural network is a perceptron which accomplishes simple signal processing, and these are then connected into a large mesh network.

The computer with the neural network is taught to do a task by having it analyze training examples, which have been previously labeled in advance. A common example of a task for a neural network using deep learning is an object recognition task, where the neural network is presented with a large number of objects of a certain type, such as a cat, or a street sign, and the computer, by analyzing the recurring patterns in the presented images, learns to categorize new images.

## ****Neural Network Architectures****

## Single Layer Perceptron (SLP)

The simplest type of perceptron has a single layer of weights connecting the inputs and output.  In this way, it can be considered the simplest kind of feed-forward network. In a feed forward network, the information always moves in one direction; it never goes backwards

Above figure shows a single-layer perceptron for easier conceptual grounding and clarification into multilayer perceptron (explained ahead). Single layer perceptron represents the m weights that are seen as a set of synapses or connecting links between one layer and another layer within the network. This parameter indicates how important each feature  [eq 10.5](https://www.analyticsvidhya.com/wp-content/uploads/2016/07/eq-10.5.png) is. Below is the adder function of features of the input multiplied by their respective synaptic connection:

[](https://www.analyticsvidhya.com/wp-content/uploads/2016/07/Eq-10.png)

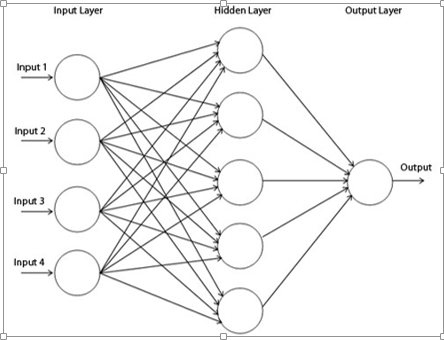
The bias[eq 10.1](https://www.analyticsvidhya.com/wp-content/uploads/2016/07/eq-10.1.png) , acts as an affine transformation to the output of the adder function [eq 10.2](https://www.analyticsvidhya.com/wp-content/uploads/2016/07/eq-10.2.png)  giving  [eq 10.3](https://www.analyticsvidhya.com/wp-content/uploads/2016/07/eq-10.3.png) , the induced local field as:

[eq 10.4](https://www.analyticsvidhya.com/wp-content/uploads/2016/07/eq-10.4.png)

## Multilayer Perceptron (MLP)

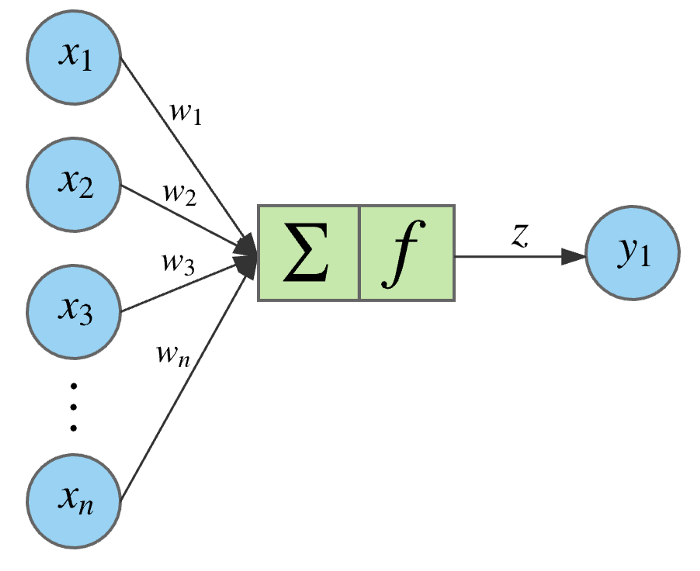
Moving onwards, multi-layer perceptron, also known as feed-forward neural networks, consists of a sequence of layers each fully connected to the next one.

A multilayer perceptron (MLP) has one or more hidden layers along with the input and output layers, each layer contains several neurons that interconnect with each other by weight links. The number of neurons in the input layer will be the number of attributes in the dataset, neurons in the output layer will be the number of classes given in the dataset.



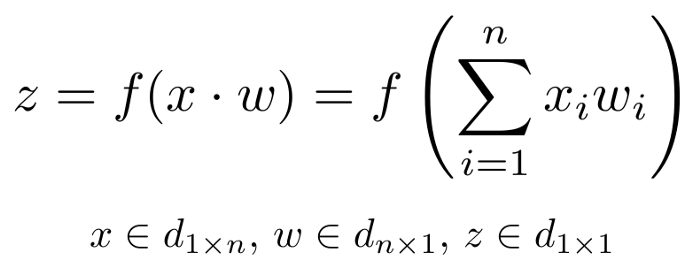
* Artificial Neural Networks

Artificial Neural Networks (ANN) are multi-layer fully-connected neural nets that look like the figure below. They consist of an input layer, multiple hidden layers, and an output layer. Every node in one layer is connected to every other node in the next layer. We make the network deeper by increasing the number of hidden layers.



A given node takes the weighted sum of its inputs, and passes it through a non-linear activation function. This is the output of the node, which then becomes the input of another node in the next layer. The signal flows from left to right, and the final output is calculated by performing this procedure for all the nodes. Training this deep neural network means learning the weights associated with all the edges.

The equation for a given node looks as follows. The weighted sum of its inputs passed through a non-linear activation function. It can be represented as a vector dot product, where n is the number of inputs for the node.



Use ANN For:

* Tabular datasets
* Classification prediction problems
* Regression prediction problems
* Recurrent Neural Network

Recurrent Neural Networks, or RNNs, were designed to work with sequence prediction problems.

Sequence prediction problems come in many forms and are best described by the types of inputs and outputs supported.

Some examples of sequence prediction problems include:

* **One-to-Many**: An observation as input mapped to a sequence with multiple steps as an output.
* **Many-to-One**: A sequence of multiple steps as input mapped to class or quantity prediction.
* **Many-to-Many**: A sequence of multiple steps as input mapped to a sequence with multiple steps as output.

The Many-to-Many problem is often referred to as sequence-to-sequence, or seq2seq for short. For more details on the types of sequence prediction problems, Recurrent neural networks were traditionally difficult to train.

The Long Short-Term Memory, or LSTM, network is perhaps the most successful RNN because it overcomes the problems of training a recurrent network and in turn has been used on a wide range of applications.

RNNs in general and LSTMs in particular have received the most success when working with sequences of words and paragraphs, generally called natural language processing.

This includes both sequences of text and sequences of spoken language represented as a time series. They are also used as generative models that require a sequence output, not only with text, but on applications such as generating handwriting.

**Use RNNs For:**

* Text data
* Speech data
* Classification prediction problems
* Regression prediction problems
* Generative models

## Convolutional Neural Networks

Convolutional Neural Networks, or CNNs, were designed to map image data to an output variable.

They have proven so effective that they are the go-to method for any type of prediction problem involving image data as an input.

The benefit of using CNNs is their ability to develop an internal representation of a two-dimensional image. This allows the model to learn position and scale in variant structures in the data, which is important when working with images.

Use CNNs For:

* Image data
* Classification prediction problems
* Regression prediction problems

More generally, CNNs work well with data that has a spatial relationship.

The CNN input is traditionally two-dimensional, a field or matrix, but can also be changed to be one-dimensional, allowing it to develop an internal representation of a one-dimensional sequence.

This allows the CNN to be used more generally on other types of data that has a spatial relationship. For example, there is an order relationship between words in a document of text. There is an ordered relationship in the time steps of a time series.

Although not specifically developed for non-image data, CNNs achieve state-of-the-art results on problems such as document classification used in sentiment analysis and related problems.

### **What is Computer Vision?**

Computer Vision is the broad parent name for any computations involving visual content – that means images, videos, icons, and anything else with pixels involved. But within this parent idea, there are a few specific tasks that are core building blocks:

* In **object classification**, you train a model on a dataset of specific objects, and the model classifies new objects as belonging to one or more of your training categories.
* For **object identification**, your model will recognize a specific instance of an object – for example, parsing two faces in an image and tagging one as Tom Cruise and one as Katie Holmes.

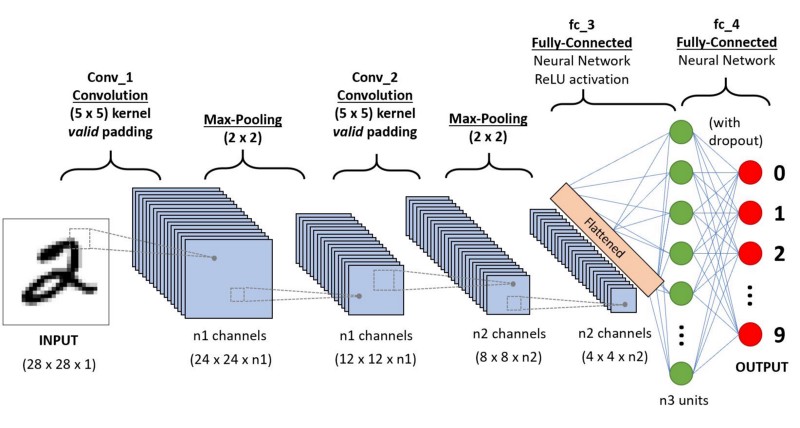
A classical application of computer vision is handwriting recognition for digitizing handwritten content (we’ll explore more use cases below). Outside of just recognition, other methods of analysis include:

* Video **motion analysis** uses computer vision to estimate the velocity of objects in a video, or the camera itself.
* In **image segmentation**, algorithms partition images into multiple sets of views.
* **Scene reconstruction**creates a 3D model of a scene inputted through images or video.
* In **image restoration**, noise such as blurring is removed from photos using Machine Learning based filters.

Any other application that involves understanding pixels through software can safely be labeled as computer vision.

* **How Computer Vision Works with 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.



## 

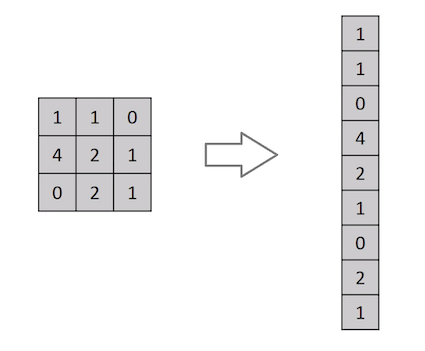
The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area.

* Why ConvNets over Feed-Forward Neural Nets?

An image is nothing but a matrix of pixel values, right? So why not just flatten the image (e.g. 3x3 image matrix into a 9x1 vector) and feed it to a Multi-Level Perceptron for classification purposes? Uh.. not really.

In cases of extremely basic binary images, the method might show an average precision score while performing prediction of classes but would have little to no accuracy when it comes to complex images having pixel dependencies throughout.

A ConvNet is able to **successfully capture the Spatial and Temporal dependencies** in an image through the application of relevant filters. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In other words, the network can be trained to understand the sophistication of the image better.



#### https://cdn-images-1.medium.com/max/600/1*15yDvGKV47a0nkf5qLKOOQ.pngInput Image

In the figure, we have an RGB image which has been separated

by its three color planes — Red, Green, and Blue. There are a

number of such color spaces in which images exist —

 Grayscale, RGB, HSV, CMYK, etc.

You can imagine how computationally intensive things would

get once the images reach dimensions, say 8K (7680×4320).

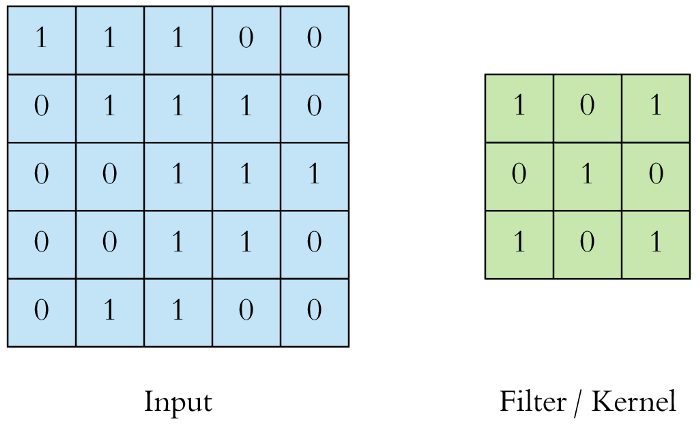
The role of the ConvNet is to reduce the images into a form

which is easier to process, without losing features which are

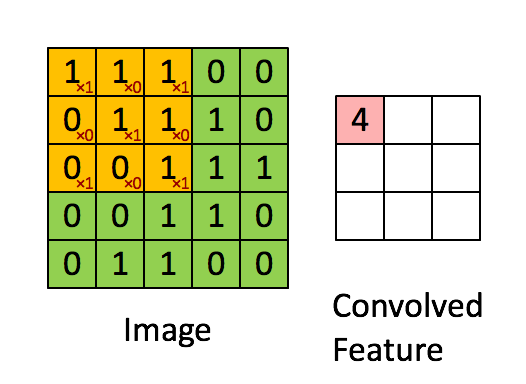
critical for getting a good prediction. This is important when we are to design an architecture which is not only good at learning features but also is scalable to massive datasets.

#### Convolution Layer — The Kernel

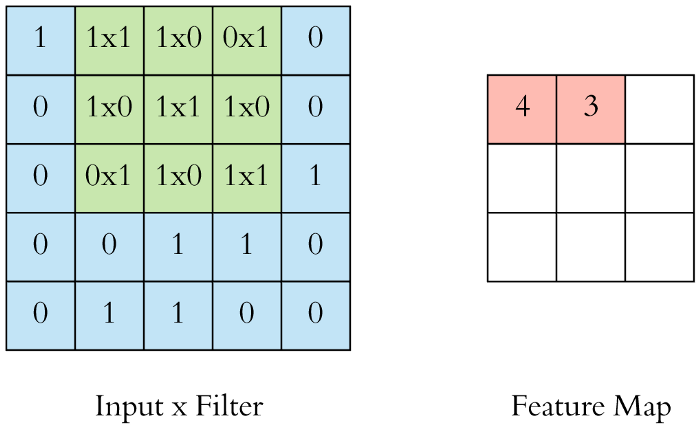
#### The main building block of CNN is the convolution layer. Convolution is a mathematical operation to merge two sets of information. In our case the convolution is applied on the input data using a convolution filter to produce a feature map. There are a lot of terms being used so let’s visualize them one by one



On the left side is the input to the convolution layer, for example the input image. On the right is the convolution filter, also called the kernel, we will use these terms interchangeably. This is called a 3x3 convolution due to the shape of the filter.We perform the convolution operation by sliding this filter over the input. At every location, we do element-wise matrix multiplication and sum the result. This sum goes into the feature map. The green area where the convolution operation takes place is called the receptive field. Due to the size of the filter the receptive field is also 3x3.

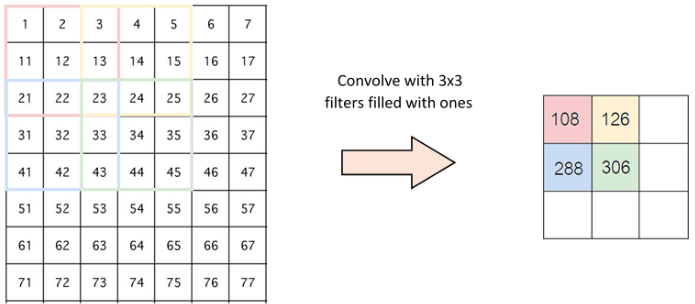


Here the filter is at the top left, the output of the convolution operation “4” is shown in the resulting feature map. We then slide the filter to the right and perform the same operation, adding that result to the feature map as well.

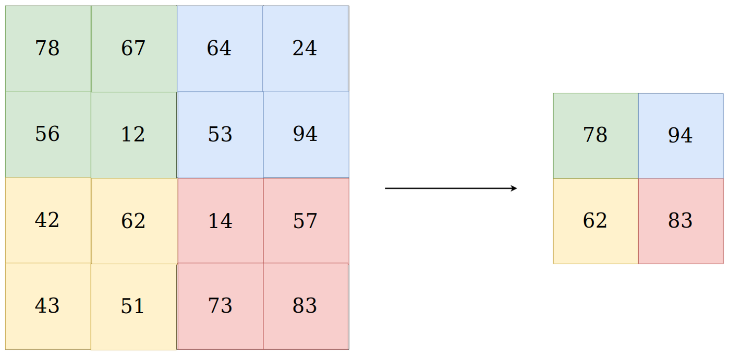


* **Strides**

Stride is the number of pixels shifts over the input matrix. When the stride is 1 then we move the filters to 1 pixel at a time. When the stride is 2 then we move the filters to 2 pixels at a time and so on. The below figure shows convolution would work with a stride of 2.

****

### Pooling Layer



There are two types of pooling:  
1) Max Pooling  
2) Average Pooling

The main purpose of a pooling layer is to reduce the number of parameters of the input tensor and thus  
- Helps reduce over fitting  
- Extract representative features from the input tensor  
- Reduces computation and thus aids efficiency

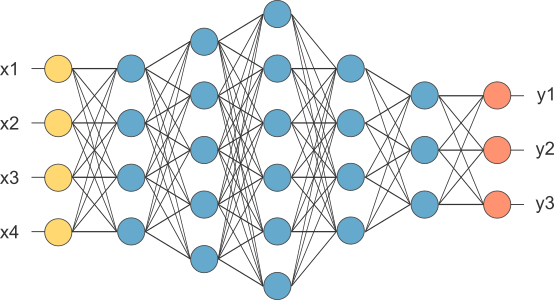
The input to the Pooling layer is tensor.

In case of Max Pooling, an example of which is shown in the Fig 6, a kernel of size n\*n (2x2 in the above example) is moved across the matrix and for each position the **max value is taken** and put in the corresponding position of the output matrix.

In case of Average Pooling, a kernel of size n\*n is moved across the matrix and for each position the **average is taken of all the values**and put in the corresponding position of the output matrix.

This is repeated for each channel in the input tensor. And so we get the output tensor.  
So, a thing to note is, **Pooling down samples the image in its height and width but the number of channels (depth) stays the same.**

* **Fully Connected Layer**



Adding a Fully-Connected layer is a (usually) cheap way of learning non-linear combinations of the high-level features as represented by the output of the convolutional layer. The Fully-Connected layer is learning a possibly non-linear function in that space.

Now that we have converted our input image into a suitable form for our Multi-Level Perceptron, we shall flatten the image into a column vector. The flattened output is fed to a feed-forward neural network and back propagation applied to every iteration of training. Over a series of epochs, the model is able to distinguish between dominating and certain low-level features in images and classify them using the **Softmax Classification** technique.

There are various architectures of CNNs available which have been key in building algorithms which power and shall power AI as a whole in the foreseeable future. Some of them have been listed below:

1. LeNet
2. AlexNet
3. VGGNet
4. GoogLeNet
5. ResNet

* **Activation Function:**
* Introduction

Internet provides access to plethora of information today. Whatever we need is just a Google (search) away. However, when we have so much of information, the challenge is to segregate between relevant and irrelevant information.

When our brain is fed with a lot of information simultaneously, it tries hard to understand and classify the information between useful and not-so-useful information. We need a similar mechanism to classify incoming information as useful or less-useful in case of Neural Networks.

This is a very important in the way a network learns because not all information is equally useful. Some of it is just noise. Well, activation functions help the network do this segregation. They help the network use the useful information and suppress the irrelevant data points.

Let us go through these activation functions, how they work and figure out which activation functions fits well into what kind of  problem statement.

* What is an Activation Function?

Activation functions are an extremely important feature of the artificial neural networks. They basically decide whether a neuron should be activated or not. Whether the information that the neuron is receiving is relevant for the given information or should it be ignored.

https://s3-ap-south-1.amazonaws.com/av-blog-media/wp-content/uploads/2017/10/17123344/act.png

The activation function is the non linear transformation that we do over the input signal. This transformed output is then sent to the next layer of neurons as input.

* Popular types of activation functions and when to use them

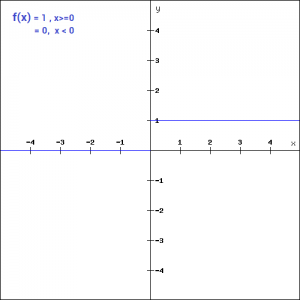
Binary Step Function

The first thing that comes to our mind when we have an activation function would be a threshold based classifier i.e. whether or not the neuron should be activated. If the value Y is above a given threshold value then activate the neuron else leave it deactivated.

It is defined as –

f(x) = 1, x>=0

= 0, x<0

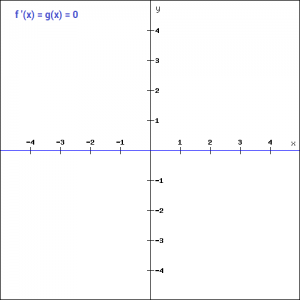
****

The binary function is extremely simple. It can be used while creating a binary classifier. When we simply need to say yes or no for a single class, step function would be the best choice, as it would either activate the neuron or leave it to zero.

The function is more theoretical than practical since in most cases we would be classifying the data into multiple classes than just a single class. The step function would not be able to do that.

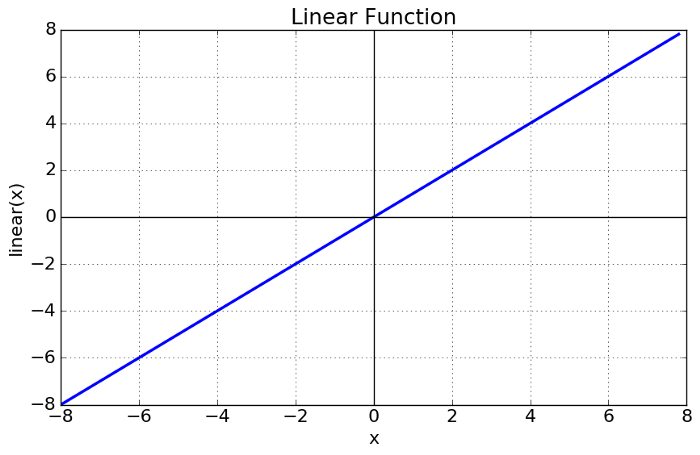
Moreover, the gradient of the step function is zero. This makes the step function not so useful since during back-propagation when the gradients of the activation functions are sent for error calculations to improve and optimize the results. The gradient of the step function reduces it all to zero and improvement of the models doesn’t really happen.

f '(x) = 0, for all x

****

1. Linear Function

* **Equation :**Linear function has the equation similar to as of a straight line i.e. **y = ax**
* No matter how many layers we have, if all are linear in nature, the final activation function of last layer is nothing but just a linear function of the input of first layer.
* **Range :** -inf to +inf
* **Uses :**Linear activation function is used at just one place i.e. output layer.
* **Issues :**If we will differentiate linear function to bring non-linearity, result will no more depend on *input “x”* and function will become constant, it won’t introduce any ground-breaking behavior to our algorithm

****

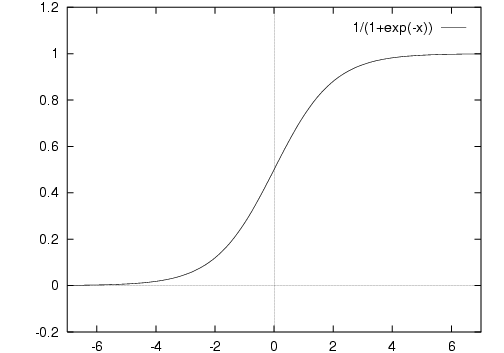
**For example :** Calculation of price of a house is a regression problem. House price may have any big/small value, so we can apply linear activation at output layer. Even in this case neural net must have any non-linear function at hidden layers.

1. Sigmoid

Sigmoid is a widely used activation function.

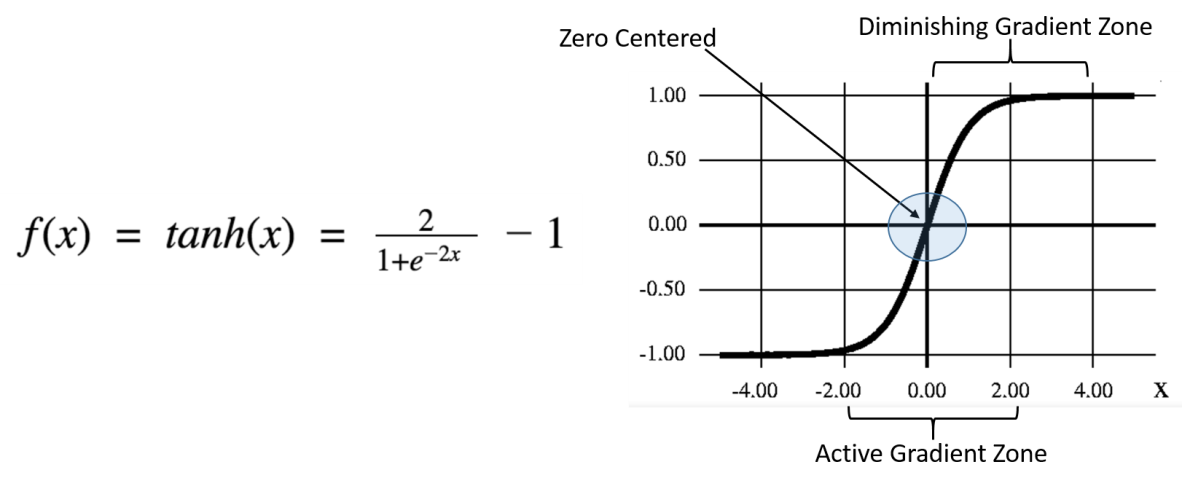
It is a function which is plotted as **‘S’** shaped graph.

* **Equation :**  
   A = 1/(1 + e-x)
* **Nature :** Non-linear. Notice that X values lies between -2 to 2, Y values are very steep. This means, small changes in x would also bring about large changes in the value of Y.
* **Value Range :**0 to 1
* **Uses :**Usually used in output layer of a binary classification, where result is either 0 or 1, as value for sigmoid function lies between 0 and 1 only so, result can be predicted easily to be ***1*** if value is greater than **0.5** and ***0*** otherwise.



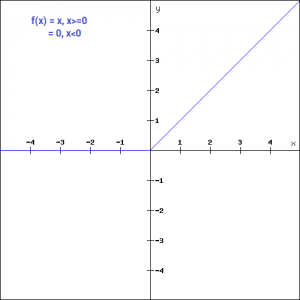
* Vanishing gradient problem
* Secondly , its output isn’t zero centered. It makes the gradient updates go too far in different directions. **0 < output < 1, and it makes optimization harder.**
* Sigmoid saturate and kill gradients.
* Sigmoid have slow convergence.

1. Tanh

* **Tanh Function :-**The activation that works almost always better than sigmoid function is Tanh function also knows as **Tangent Hyperbolic function**. It’s actually mathematically shifted version of the sigmoid function. Both are similar and can be derived from each other.
* **Equation :-**
* f(x) = tanh(x) = 2/(1 + e-2x) - 1
* OR
* tanh(x) = 2 \* sigmoid(2x) - 1
* **Value Range :-**-1 to +1
* **Nature :-**non-linear
* **Uses :-**Usually used in hidden layers of a neural network as it’s values lies between **-1 to 1**hence the mean for the hidden layer comes out be 0 or very close to it, hence helps in centering the data by bringing mean close to 0. This makes learning for the next layer much easier.

1. ReLU

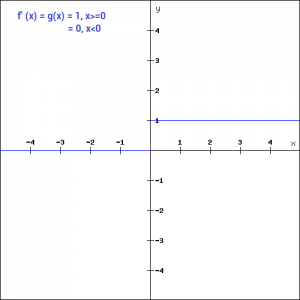
* **RELU :-**Stands for *Rectified linear unit*. It is the most widely used activation function. Chiefly implemented in *hidden layers* of Neural network.
* **Equation :- *A(x) = max(0,x)***. It gives an output x if x is positive and 0 otherwise.
* **Value Range :-**[0, inf)
* **Nature :-**non-linear, which means we can easily back propagate the errors and have multiple layers of neurons being activated by the ReLU function.
* **Uses :-**ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. At a time only a few neurons are activated making the network sparse making it efficient and easy for computation

**

ReLU is the most widely used activation function while designing networks today. First things first, the ReLU function is nonlinear, which means we can easily backpropagate the errors and have multiple layers of neurons being activated by the ReLU function.

The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time. What does this mean? If you look at the ReLU function if the input is negative it will convert it to zero and the neuron does not get activated. This means that at a time only a few neurons are activated making the network sparse making it efficient and easy for computation.

Let’s look at the gradient of the ReLU function.



But ReLU also falls a prey to the gradients moving towards zero. If you look at the negative side of the graph, the gradient is zero, which means for activations in that region, the gradient is zero and the weights are not updated during back propagation. This can create dead neurons which never get activated. When we have a problem, we can always engineer a solution.

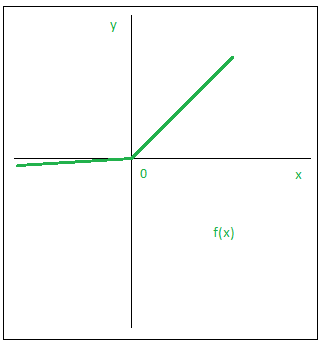
1. Leaky ReLU

Leaky ReLU function is nothing but an improved version of the ReLU function. As we saw that for the ReLU function, the gradient is 0 for x<0, which made the neurons die for activations in that region. Leaky ReLU is defined to address this problem. Instead of defining the ReLU function as 0 for x less than 0, we define it as a small linear component of x. It can be defined as-

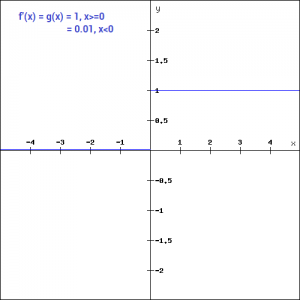
f(x)= ax, x<0

= x, x>=0

What we have done here is that we have simply replaced the horizontal line with a non-zero, non-horizontal line. Here is a small value like 0.01 or so. It can be represented on the graph as-



The main advantage of replacing the horizontal line is to remove the zero gradient. So in this case the gradient of the left side of the graph is non zero and so we would no longer encounter dead neurons in that region. The gradient of the graph would look like



Similar to the Leaky ReLU function, we also have the **Parameterized ReLU function**. It is defined similar to the Leaky ReLU as –

f(x)= ax, x<0

= x, x>=0

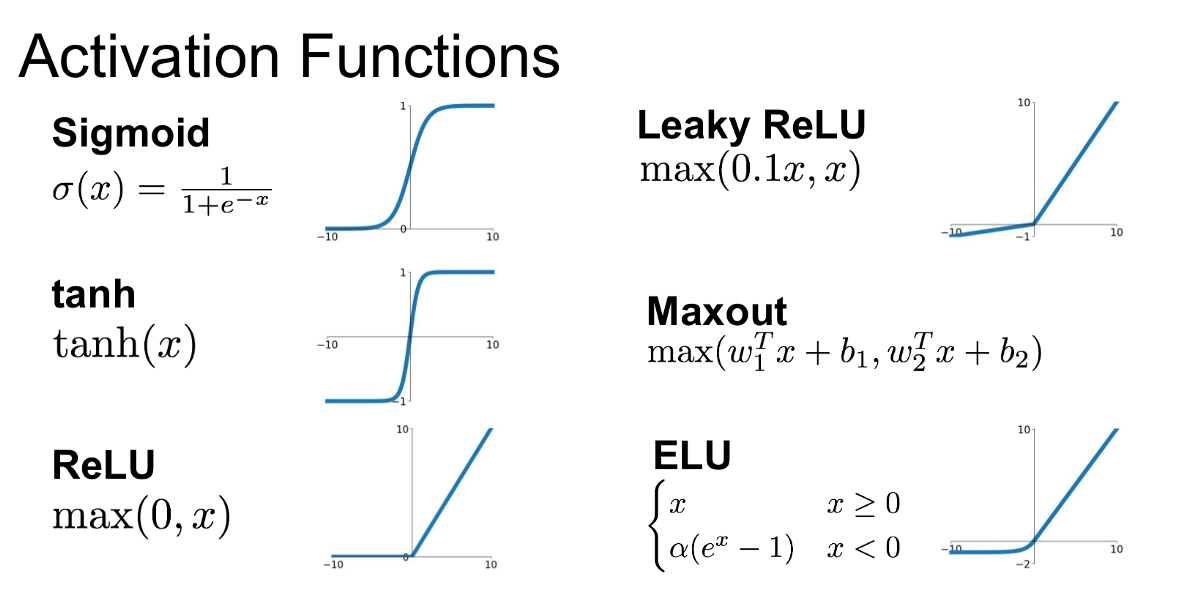
1. Softmax

* **Softmax Function :-**The softmax function is also a type of sigmoid function but is handy when we are trying to handle classification problems.
* **Nature :-**non-linear
* **Uses :-**Usually used when trying to handle multiple classes. The softmax function would squeeze the outputs for each class between 0 and 1 and would also divide by the sum of the outputs.
* **Ouput:-**The softmax function is ideally used in the output layer of the classifier where we are actually trying to attain the probabilities to define the class of each input.
* The softmax function is ideally used in the output layer of the classifier where we are actually trying to attain the probabilities to define the class of each input.
* Choosing the right Activation Function

Now that we have seen so many activation functions, we need some logic / heuristics to know which activation function should be used in which situation. Good or bad – there is no rule of thumb.

However depending upon the properties of the problem we might be able to make a better choice for easy and quicker convergence of the network.

* Sigmoid functions and their combinations generally work better in the case of classifiers
* Sigmoid and tanh functions are sometimes avoided due to the vanishing gradient problem
* ReLU function is a general activation function and is used in most cases these days
* If we encounter a case of dead neurons in our networks the leaky ReLU function is the best choice
* Always keep in mind that ReLU function should only be used in the hidden layers
* As a rule of thumb, you can begin with using ReLU function and then move over to other activation functions in case ReLU doesn’t provide with optimum results.



* **Optimizers**

# **Role of an optimizer**

Optimizers update the weight parameters to minimize the loss function. Loss function acts as guides to the terrain telling optimizer if it is moving in the right direction to reach the bottom of the valley, the global minimum.

1. Gradient Descent

Gradient descent is an iterative machine learning optimization algorithm to reduce the cost function. This will help models to make accurate predictions.

Gradient indicates the direction of increase. As we want to find the minimum point in the valley we need to go in the opposite direction of the gradient. We update parameters in the negative gradient direction to minimize the loss.

Different types of Gradient descents are

* Batch Gradient Descent or Vanilla Gradient Descent
* Stochastic Gradient Descent
* Mini batch Gradient Descent

### Stochastic gradient descent

Stochastic Gradient Descent(SGD) on the other hand performs a parameter update for each training example .It is usually much faster technique.It performs one update at a time.

θ=θ−η⋅∇J

Now due to these frequent updates ,parameters updates have high variance and causes the Loss function to fluctuate to different intensities. This is actually a good thing because it helps us discover new and possibly better local minima , whereas Standard Gradient Descent will only converge to the minimum of the basin as mentioned above.

But the problem with SGD is that due to the frequent updates and fluctuations it ultimately complicates the convergence to the exact minimum and will keep overshooting due to the frequent fluctuation

The problems of high variance parameter updates and unstable convergence can be rectified in another variant called Mini-Batch Gradient Descent.

keras.optimizers.SGD(lr=0.01, momentum=0.0, decay=0.0, nesterov=**False**)

Stochastic gradient descent optimizer.

Includes support for momentum, learning rate decay, and Nesterov momentum.

**Arguments**

* **lr**: float >= 0. Learning rate.
* **Momentum**: float >= 0. Parameter that accelerates SGD in the relevant direction and dampens oscillations.
* **Decay**: float >= 0. Learning rate decay over each update.
* **Nesterov**: Boolean. Whether to apply Nesterov momentum.

1. Mini Batch Gradient Descent

An improvement to avoid all the problems and demerits of SGD and standard Gradient Descent would be to use **Mini Batch Gradient Descent** as it takes the best of both techniques and performs an update for every batch with n training examples in each batch.

The advantages of using Mini Batch Gradient Descent are —

* It Reduces the variance in the parameter updates , which can ultimately lead us to a much better and stable convergence.
* Can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient.
* Commonly Mini-batch sizes Range from 50 to 256, but can vary as per the application and problem being solved.
* Mini-batch gradient descent is typically the algorithm of choice when training a neural network nowadays

### RMSprop

It is recommended to leave the parameters of this optimizer at their default values (except the learning rate, which can be freely tuned).

This optimizer is usually a good choice for recurrent neural networks.

keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=**None**, decay=0.0)

**Arguments**

* **lr**: float >= 0. Learning rate.
* **rho**: float >= 0.
* **epsilon**: float >= 0. Fuzz factor. If None, defaults to K.epsilon().
* **decay**: float >= 0. Learning rate decay over each update.

### Adagrad

It simply allows the learning Rate -**η**to **adapt** based on the parameters. So it makes big updates for infrequent parameters and small updates for frequent parameters. For this reason, it is well-suited for dealing with sparse data.

Adagrad is an optimizer with parameter-specific learning rates, which are adapted relative to how frequently a parameter gets updated during training. The more updates a parameter receives, the smaller the learning rate. It is recommended to leave the parameters of this optimizer at their default values. Its main weakness is that its learning rate-**η** is always Decreasing and decaying.

keras.optimizers.Adagrad(lr=0.01, epsilon=**None**, decay=0.0)

**Arguments**

* **lr**: float >= 0. Initial learning rate.
* **epsilon**: float >= 0. If None, defaults to K.epsilon().
* **decay**: float >= 0. Learning rate decay over each update.

### Adadelta

Adadelta is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients. This way, Adadelta continues learning even when many updates have been done. Compared to Adagrad, in the original version of Adadelta you don't have to set an initial learning rate. In this version, initial learning rate and decay factor can be set, as in most other Keras optimizers.

It is recommended to leave the parameters of this optimizer at their default values.

keras.optimizers.Adadelta(lr=1.0, rho=0.95, epsilon=**None**, decay=0.0)

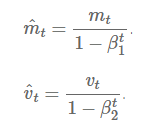
**Arguments**

* **lr**: float >= 0. Initial learning rate, defaults to 1. It is recommended to leave it at the default value.
* **rho**: float >= 0. Adadelta decay factor, corresponding to fraction of gradient to keep at each time step.
* **epsilon**: float >= 0. Fuzz factor. If None, defaults to K.epsilon().
* **decay**: float >= 0. Initial learning rate decay.

### Adam

Adam stands for **Adaptive Moment Estimation.** Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients like **AdaDelta** ,**Adam** also keeps an exponentially decaying average of past gradients M**(t)**, similar to momentum:

**M(t) and V(t)** are values of the first moment which is the **Mean** and the second moment which is the uncentered variance of the gradients respectively.



Adam works well in practice and compares favorably to other adaptive learning-method algorithms as it converges very fast and the learning speed of the Model is quiet Fast and efficient and also it rectifies every problem that is faced in other optimization techniques such as vanishing Learning rate , slow convergence or High variance in the parameter updates which leads to fluctuating Loss function

keras.optimizers.Adam(lr=0.001, beta\_1=0.9, beta\_2=0.999, epsilon=**None**, decay=0.0, amsgrad=**False**)

Default parameters follow those provided in the original paper.

**Arguments**

* **lr**: float >= 0. Learning rate.
* **beta\_1**: float, 0 < beta < 1. Generally close to 1.
* **beta\_2**: float, 0 < beta < 1. Generally close to 1.
* **epsilon**: float >= 0. Fuzz factor. If None, defaults to K.epsilon().
* **decay**: float >= 0. Learning rate decay over each update.
* **amsgrad**: Boolean. Whether to apply the AMSGrad variant of this algorithm from the paper "On the Convergence of Adam and Beyond".
* **beta\_2**: floats, 0 < beta < 1. Generally close to 1.
* **epsilon**: float >= 0. Fuzz factor. If None, defaults to K.epsilon().
* **decay**: float >= 0. Learning rate decay over each update.

### Nadam

Much like Adam is essentially RMSprop with momentum, Nadam is Adam RMSprop with Nesterov momentum.

Default parameters follow those provided in the paper. It is recommended to leave the parameters of this optimizer at their default values

keras.optimizers.Nadam(lr=0.002, beta\_1=0.9, beta\_2=0.999, epsilon=**None**, schedule\_decay=0.004)

.

**Arguments**

* **Lr**: float >= 0. Learning rate.
* **beta\_1**: floats, 0 < beta < 1. Generally close to 1.
* **beta\_2**: floats, 0 < beta < 1. Generally close to 1.
* **epsilon**: float >= 0. Fuzz factor. If None, defaults to K.epsilon().
* **schedule\_decay**: floats, 0 < schedule\_decay < 1.

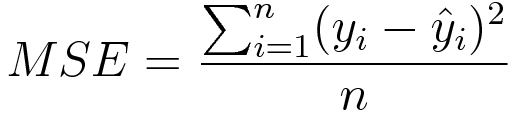
# **Loss Functions:**

Thus, loss functions are helpful to train a neural network. Given an input and a target, they calculate the loss, i.e difference between output and target variable. Loss functions fall under four major category:

* Regression Problem
* A problem where you predict a real-value quantity.
* Output Layer Configuration : One node with a linear activation unit.
* Loss Function : Mean Squared Error (MSE)
* Binary Classification Problem
* A problem where you classify an example as belonging to one of two classes.
* The problem is framed as predicting the likelihood of an example belonging to class one, e.g. the class that you assign the integer value 1, whereas the other class is assigned the value 0.
* Output Layer Configuration : One node with a sigmoid activation unit.
* Loss Function : Binary Cross-Entropy, also referred to as Logarithmic loss.
* Multi-Class Classification Problem
* A problem where you classify an example as belonging to one of more than two classes.
* The problem is framed as predicting the likelihood of an example belonging to each class.
* Output Layer Configuration : One node for each class using the softmax activation function.
* Loss Function : Cross-Entropy, also referred to as Logarithmic loss.

### Mean Squared Error Loss

Mean Squared Error loss, or MSE for short, is calculated as the average of the squared differences between the predicted and actual values.

The result is always positive regardless of the sign of the predicted and actual values and a perfect value is 0.0. The loss value is minimized, although it can be used in a maximization optimization process by making the score negative.

### Binary Cross-Entropy Loss

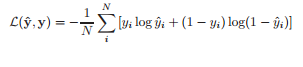
[Cross-entropy](https://en.wikipedia.org/wiki/Cross_entropy) is the default loss function to use for binary classification problems.

It is intended for use with binary classification where the target values are in the set {0, 1}

Mathematically, it is the preferred loss function under the inference framework of maximum likelihood. It is the loss function to be evaluated first and only changed if you have a good reason.

Cross-entropy will calculate a score that summarizes the average difference between the actual and predicted probability distributions for predicting class 1. The score is minimized and a perfect cross-entropy value is 0.

Cross-entropy can be specified as the loss function in Keras by specifying ‘binary\_crossentropy‘ when compiling the model.

***model.compile(loss='binary\_crossentropy', optimizer=opt, metrics=['accuracy'])***

1. Multi-Class Cross-Entropy Loss

[Cross-entropy](https://en.wikipedia.org/wiki/Cross_entropy) is the default loss function to use for multi-class classification problems.

In this case, it is intended for use with multi-class classification where the target values are in the set {0, 1, 3, …, n}, where each class is assigned a unique integer value.

Mathematically, it is the preferred loss function under the inference framework of maximum likelihood. It is the loss function to be evaluated first and only changed if you have a good reason.

Cross-entropy will calculate a score that summarizes the average difference between the actual and predicted probability distributions for all classes in the problem. The score is minimized and a perfect cross-entropy value is 0.

Cross-entropy can be specified as the loss function in Keras by specifying ‘categorical\_crossentropy‘ when compiling the model.

***model.compile(loss='categorical\_crossentropy', optimizer=opt, metrics=['accuracy'])***

cc1.gif

1. **Sparse Multiclass Cross-Entropy Loss**

A possible cause of frustration when using cross-entropy with classification problems with a large number of labels is the one hot encoding process.

For example, predicting words in a vocabulary may have tens or hundreds of thousands of categories, one for each label. This can mean that the target element of each training example may require a one hot encoded vector with tens or hundreds of thousands of zero values, requiring significant memory.

Sparse cross-entropy addresses this by performing the same cross-entropy calculation of error, without requiring that the target variable be one hot encoded prior to training.

Sparse cross-entropy can be used in keras for multi-class classification by using ‘sparse\_categorical\_crossentropy‘ when calling the compile() function.

**model.compile(loss='sparse\_categorical\_crossentropy', optimizer=opt, metrics=['accuracy'])**

The function requires that the output layer is configured with an n nodes (one for each class), in this case three nodes, and a ‘softmax‘ activation in order to predict the probability for each class.

|  |  |
| --- | --- |
| 1 | ***model.add(Dense(3, activation='softmax'))*** |

1. Kullback Leibler Divergence Loss

[Kullback Leibler Divergence](https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence), or KL Divergence for short, is a measure of how one probability distribution differs from a baseline distribution.

A KL divergence loss of 0 suggests the distributions are identical. In practice, the behavior of KL Divergence is very similar to cross-entropy. It calculates how much information is lost (in terms of bits) if the predicted probability distribution is used to approximate the desired target probability distribution.

As such, the KL divergence loss function is more commonly used when using models that learn to approximate a more complex function than simply multi-class classification, such as in the case of an autoencoder used for learning a dense feature representation under a model that must reconstruct the original input. In this case, KL divergence loss would be preferred. Nevertheless, it can be used for multi-class classification, in which case it is functionally equivalent to multi-class cross-entropy.

KL divergence loss can be used in Keras by specifying ‘kullback\_leibler\_divergence‘ in the compile() function.

**model.compile(loss='kullback\_leibler\_divergence', optimizer=opt, metrics=['accuracy'])**

**Project**

* Dataset:

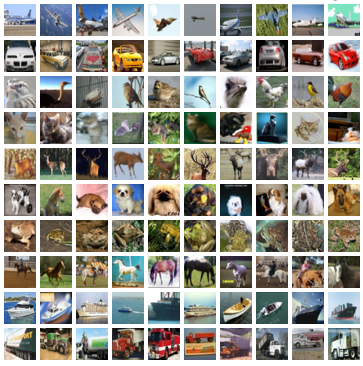
The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

* Project Objective:

For each image in the test set, predict a label for the given id. Labels must match the official labels exactly {airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck}.

* Dataset Description:

[CIFAR-10](http://www.cs.toronto.edu/~kriz/cifar.html) is an established computer-vision dataset used for object recognition. It is a subset of the [80 million tiny images dataset](http://groups.csail.mit.edu/vision/TinyImages/) and consists of 60,000 32x32 color images containing one of 10 object classes, with 6000 images per class. It was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.

Here are the classes in the dataset, as well as 10 random images from each:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Airplane** |  |  |  |  |  |  |  |  |  |  |
| **Automobile** |  |  |  |  |  |  |  |  |  |  |
| **Bird** |  |  |  |  |  |  |  |  |  |  |
| **Cat** |  |  |  |  |  |  |  |  |  |  |
| **Deer** |  |  |  |  |  |  |  |  |  |  |
| **Dog** |  |  |  |  |  |  |  |  |  |  |
| **Frog** |  |  |  |  |  |  |  |  |  |  |
| **Horse** |  |  |  |  |  |  |  |  |  |  |
| **Ship** |  |  |  |  |  |  |  |  |  |  |
| **truck** |  |  |  |  |  |  |  |  |  |  |

* Environment used:
* Google Colaboratory
* Why we use Google Colab?
* Google Colaboratory is based on the [Jupyter notebook](http://jupyter.org/) design and operation paradigm.
* The main existing deep learning frameworks like Tensor Flow, Keras and PyTorch are maturing and offer a lot of functionality to streamline the deep learning process. There are also other great tool sets emerging for the deep learning practitioner. One of these is the Google Colaboratory environment. This environment, based on Python Jupyter notebooks, gives the user free access to Tesla K80 GPUs.
* One of the most important and useful components of Google Colaboratory is the ability to share your notebooks with others, and also allow others to comment on your work.
* Keras
* Why we use Keras API?

There are countless deep learning frameworks available today. Why use Keras rather than any other? Here are some of the areas in which Keras compares favorably to existing alternatives.

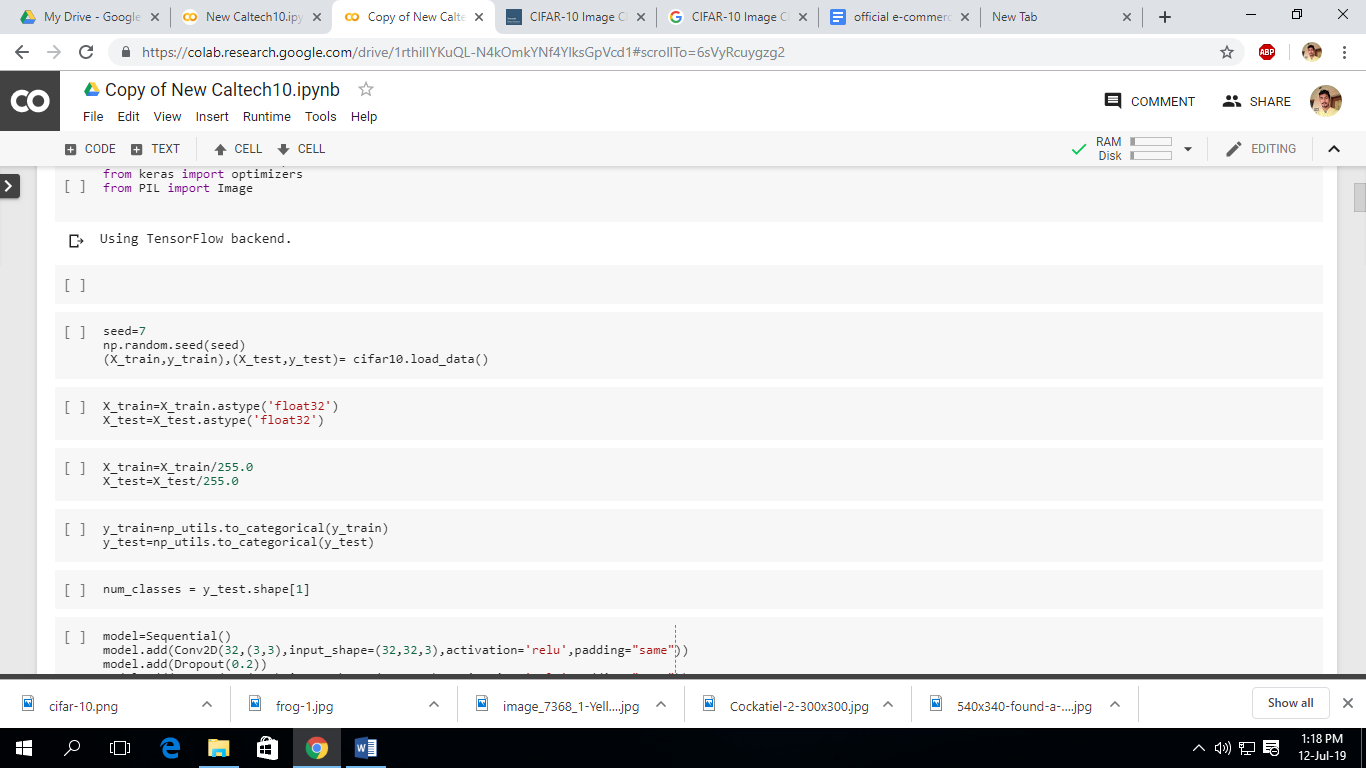
* Keras is an API designed for human beings, not machines. [Keras follows best practices for reducing cognitive load](https://blog.keras.io/user-experience-design-for-apis.html): it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear and actionable feedback upon user error.
* This makes Keras easy to learn and easy to use. As a Keras user, you are more productive, allowing you to try more ideas than your competition, faster -- which in turn [helps you win machine learning competitions](https://www.quora.com/Why-has-Keras-been-so-successful-lately-at-Kaggle-competitions).
* This ease of use does not come at the cost of reduced flexibility: because Keras integrates with lower-level deep learning languages (in particular TensorFlow), it enables you to implement anything you could have built in the base language. In particular, as tf.keras, the Keras API integrates seamlessly with your TensorFlow workflows.
* Import Packages:

At first we have to import the packages from Keras API such as models, labels, processing, utils, optimizers etc. Also Dataset imported from *Keras.datasets*.

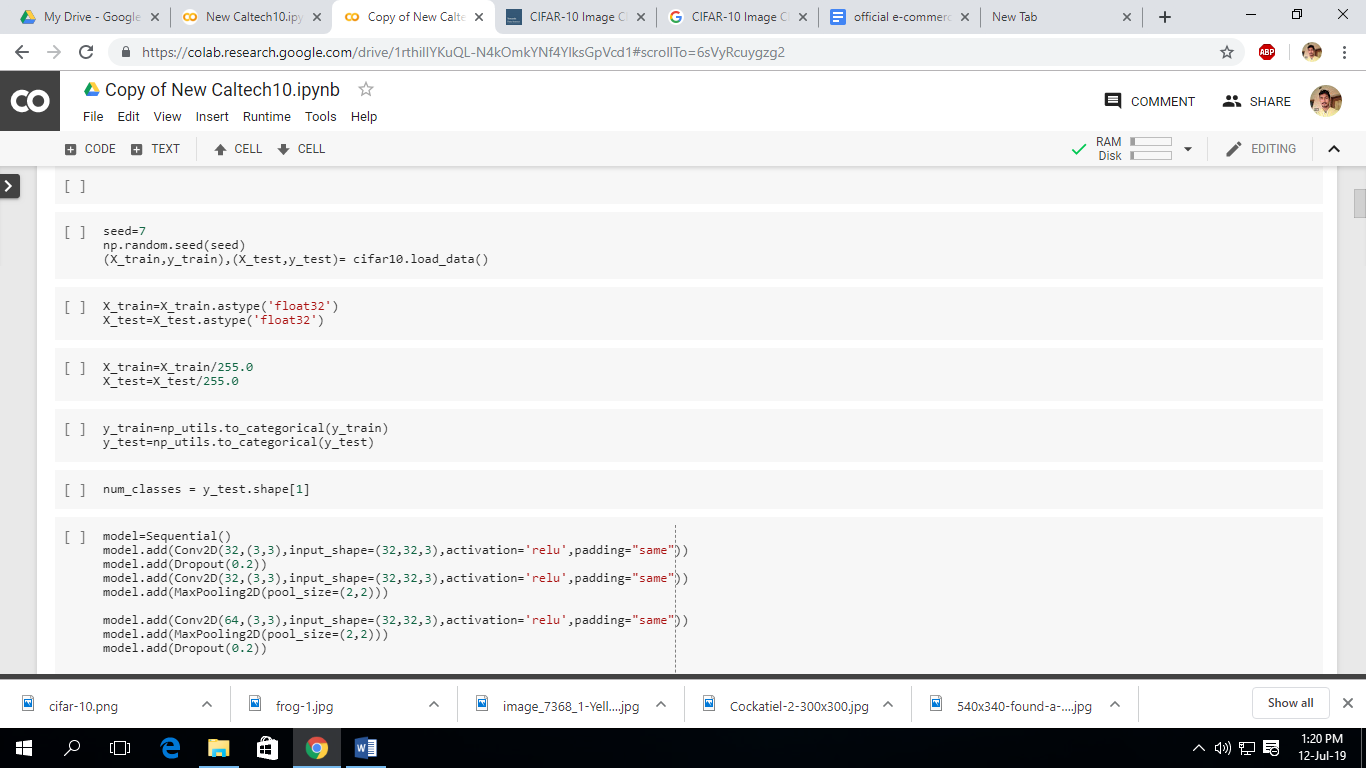
And also we import python packages for numerical computation and visualization such as numpy, pandas, matplotlib etc.



Now, dividing the dataset into two parts: train set and test set.



Scaling and classes the dataset.



* Build the model:

Convolutional networks are built from several types of layers:

* [Conv2D](https://keras.io/layers/convolutional/#conv2d) — performs convolution:
* **filters**: number of output channels;
* **kernel\_size**: an integer or tuple/list of 2 integers, specifying the width and height of the 2D convolution window;
* **padding**: padding=”same” adds zero padding to the input, so that the output has the same width and height, padding=’valid’ performs convolution only in locations where kernel and the input fully overlap;
* **activation**: “relu”, “tanh”, etc.
* **input\_shape**: shape of input.
* [MaxPooling2D](https://keras.io/layers/pooling/#maxpooling2d) — performs 2D max pooling.
* [Flatten](https://keras.io/layers/core/#flatten) — flattens the input, does not affect the batch size.
* [Dense](https://keras.io/layers/core/#dense) — fully-connected layer.
* [Dropout](https://keras.io/layers/core/#dropout) — applies dropout.

**Our**input image is a tensor whose width is 32 pixels and height is 32 pixels with 3 channels representing RGB( red, green, blue) color intensities. Thus we need to define a model which takes **(None, 32, 32, 3)** input shape and predicts **(None, 10)** output with probabilities for all classes. **None**in shapes stands for batch.

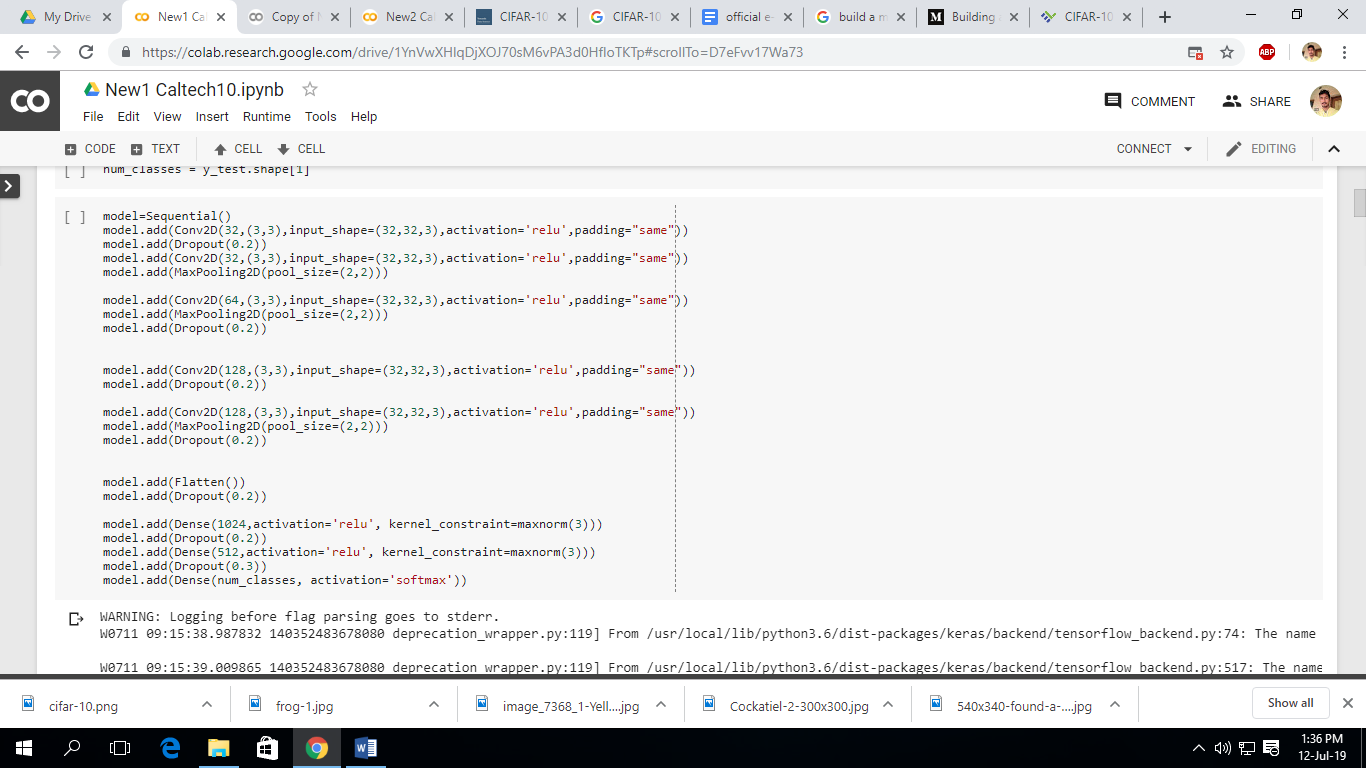
Simple feed-forward networks in Keras can be defined in the following way:

We will Stack **5** convolutional layers with kernel size **(3, 3)** with growing number of filters **(32, 32, 64, 128, 128)**with padding size of same so input and output images will have same dimensions.

We will add **2x2** pooling layer after every 2 convolutional layers (conv-conv-pool scheme).

We will use **ReLU** activation function.

After adding all conv-conv-pool scheme layers we will add a dense layer with **1024** neurons and a second dense layer with **512** neurons for classes. Remember we need to use **Flatten** layer before first dense layer to reshape input volume into a flat vector!

**Dropout** layer after every pooling layer will be used to reduce overfitting in the model with setting the fraction value to 0.2 so that 0.2 fraction of units will be dropped.

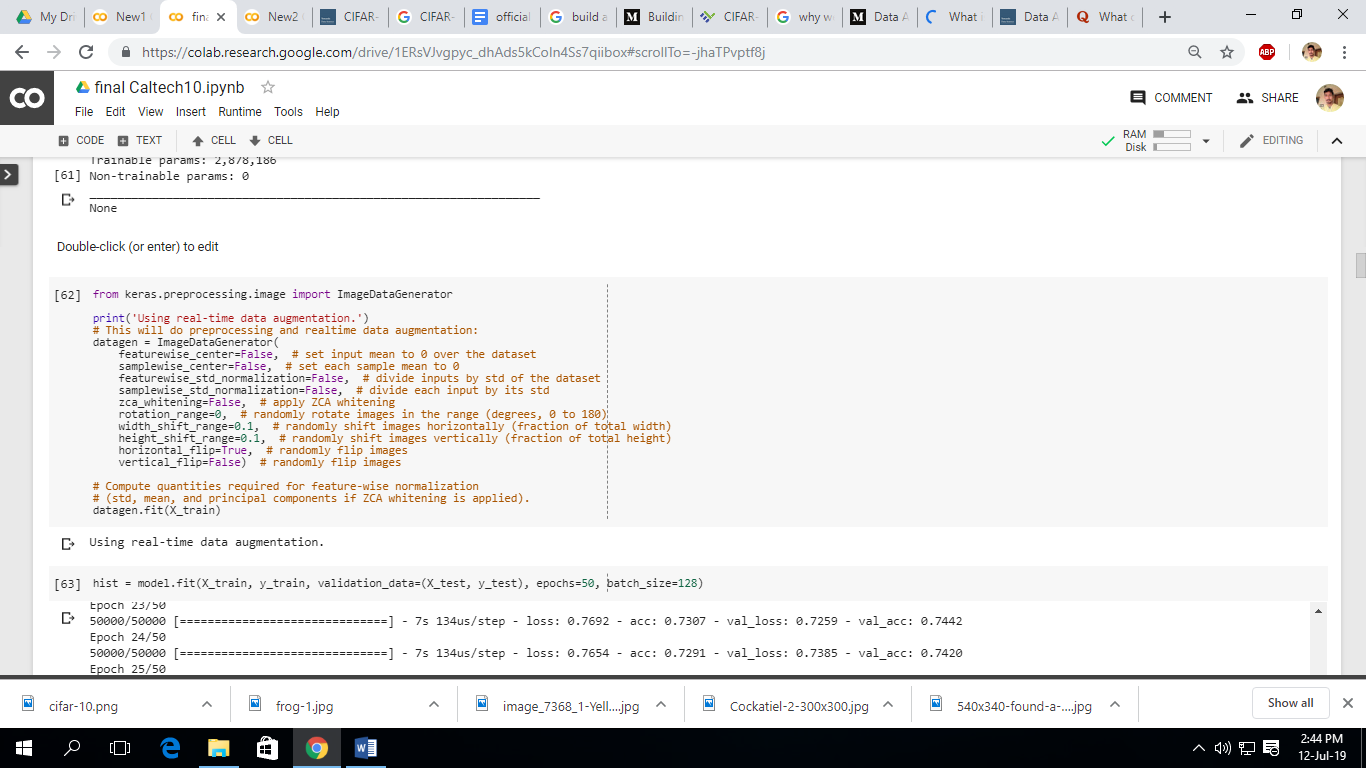
* Data Augmentation

Data augmentation means increasing the number of data points. In terms of images, it may mean that increasing the number of images in the dataset. In terms of traditional row/column format data, it means increasing the number of rows or objects.

* Why we use data augmentation?

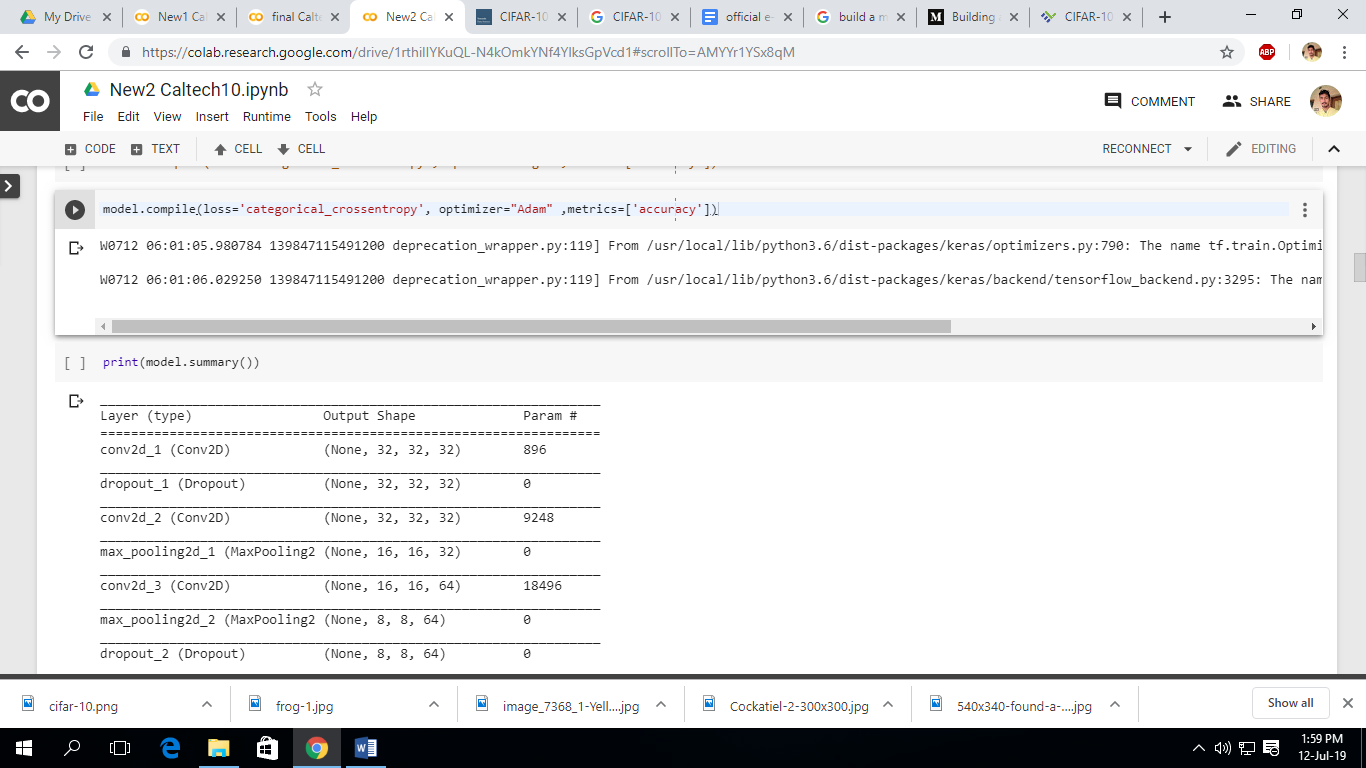
 Invariance is the ability of convolutional neural networks to classify objects even when they are placed in different orientations. **Data augmentation is a way of creating new ‘data’ with different orientations**. The benefits of this are two fold, the first being the ability to generate ‘more data’ from limited data and secondly it prevents over fitting.

Working with limited data has its own challenges, using data augmentation can have positive results only if the augmentation techniques enhance the current data set.

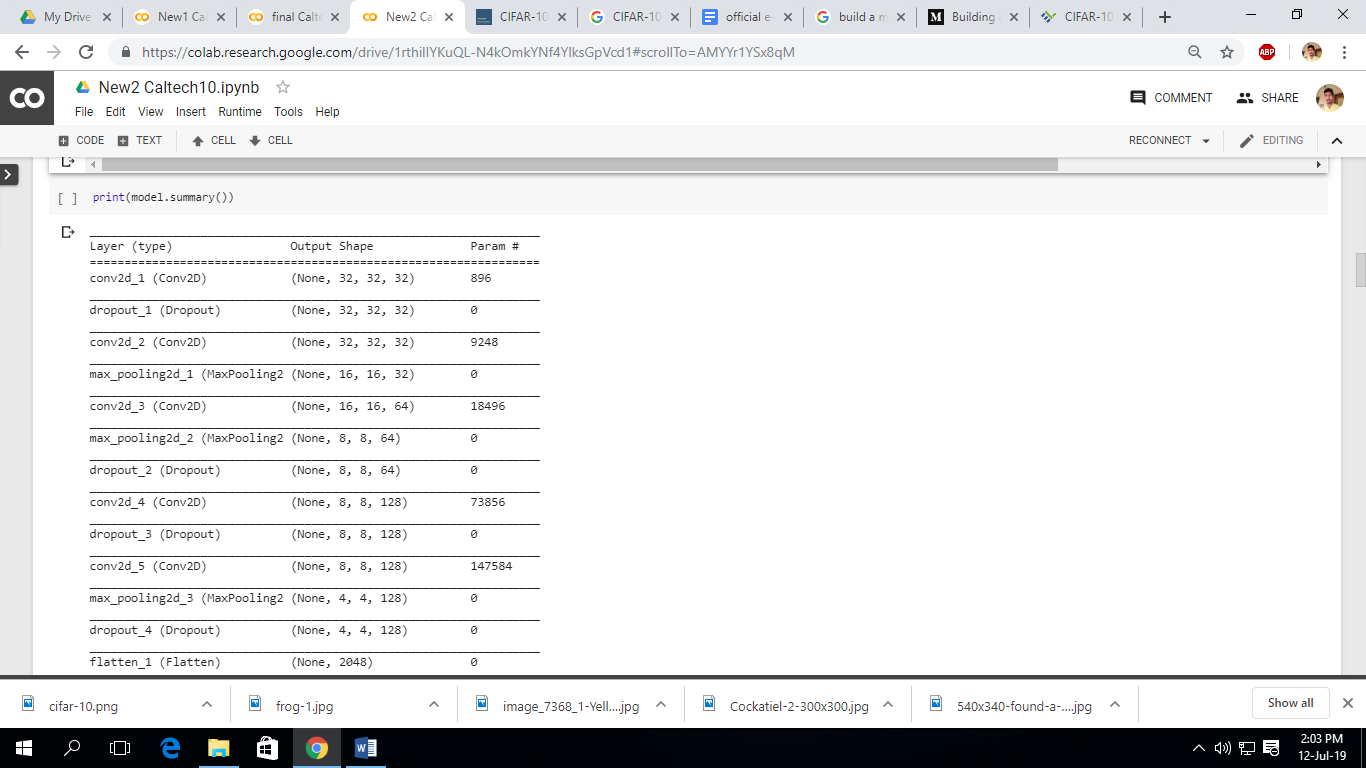
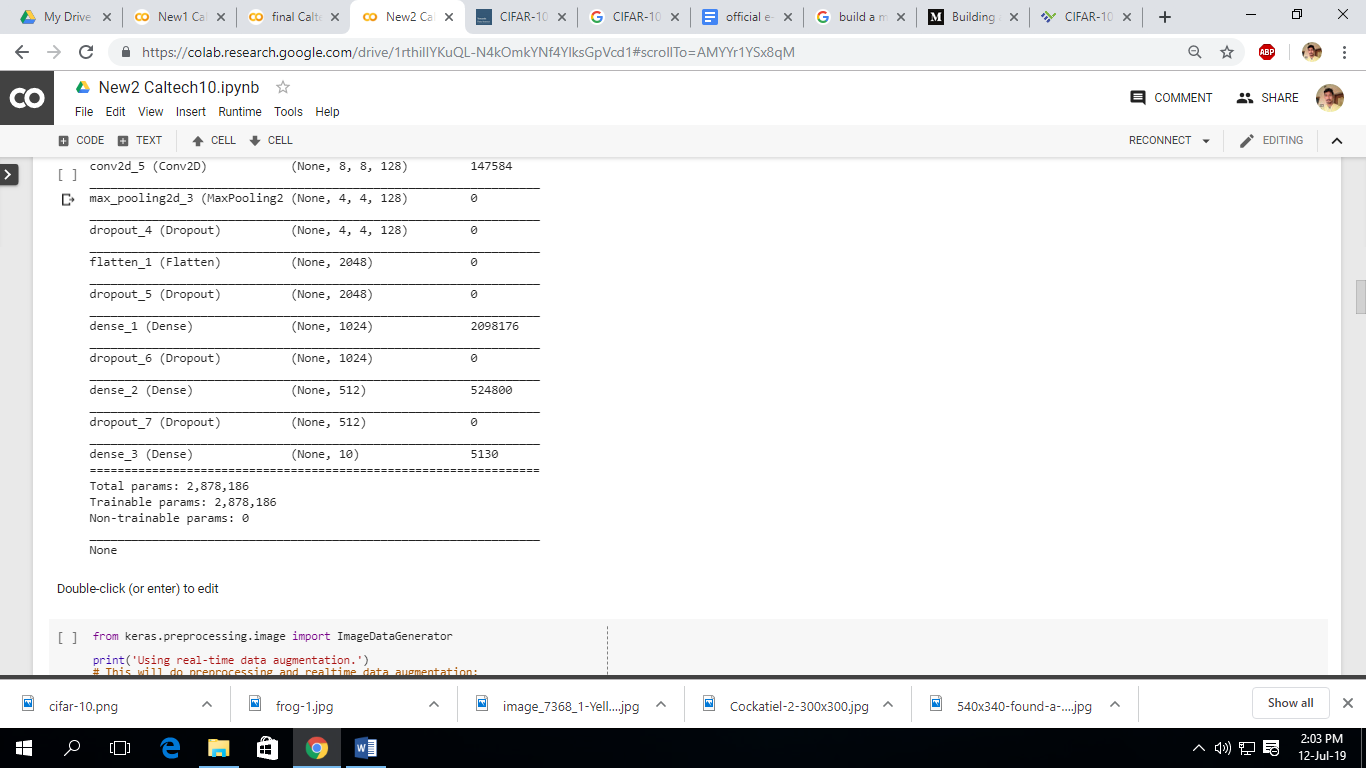


Now, we will compile the model using model.compile() and assign loss='categorical\_crossentropy' and we can use different optimizers. Here we use ‘Adam’ and ‘SGD’ and verify which one is better with respect to model’s accuracy.

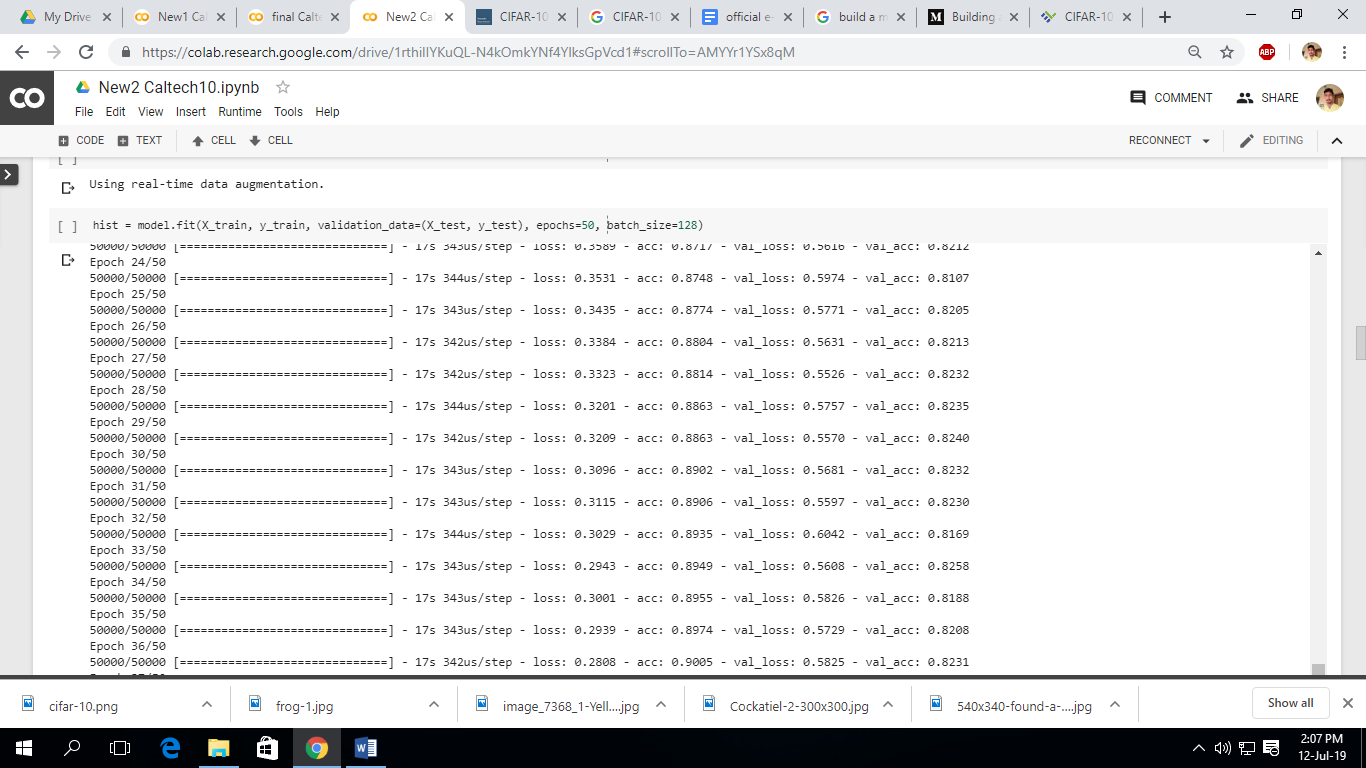
* Model using Adam

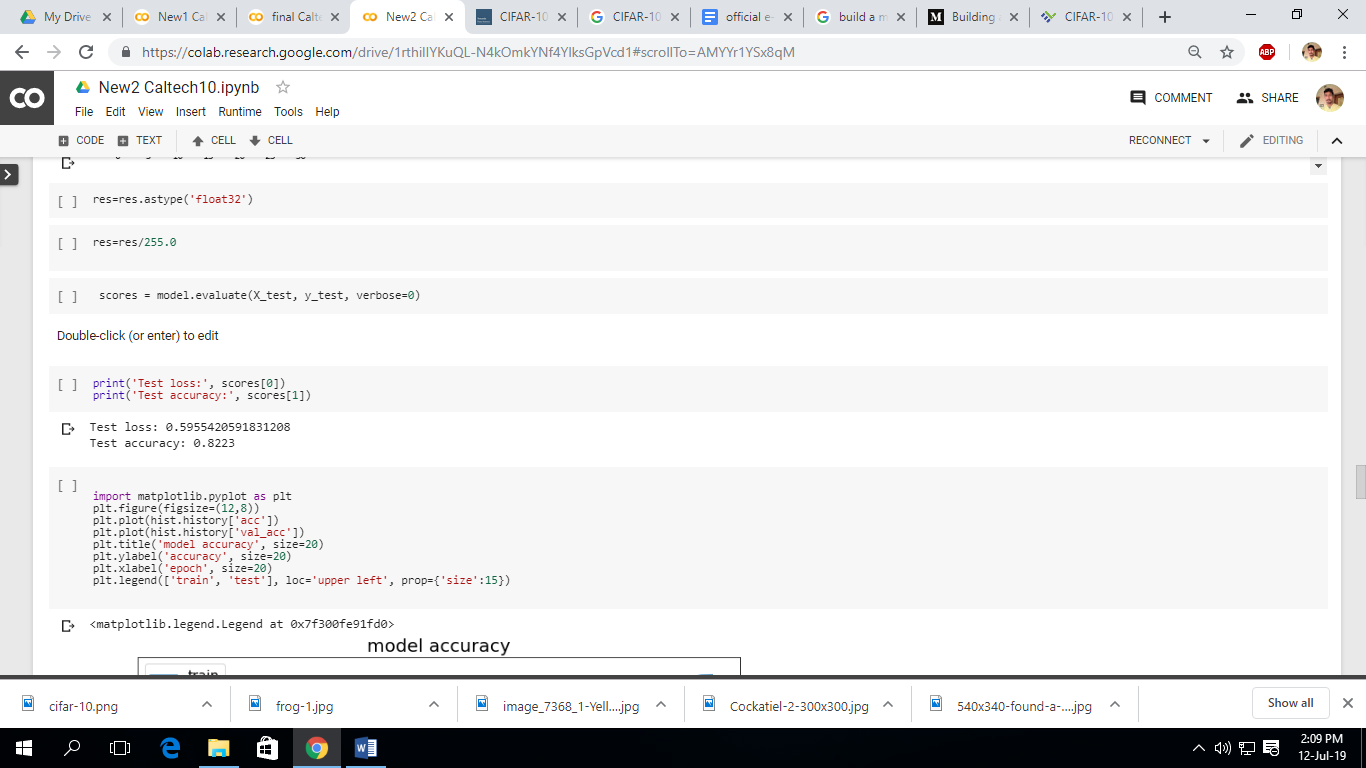


now, we print the model summary



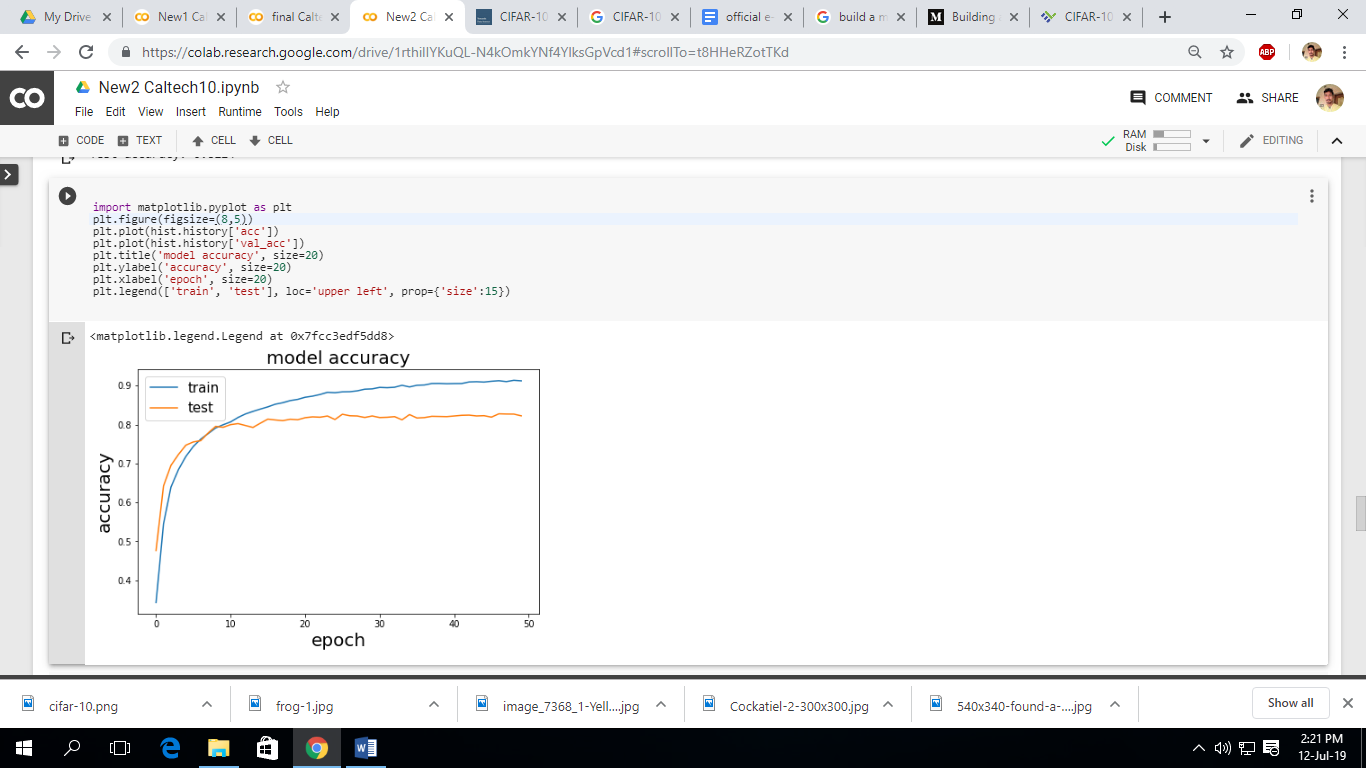
Now, we fit the model using validation and train data and we take 50 epochs and 128 batch size.



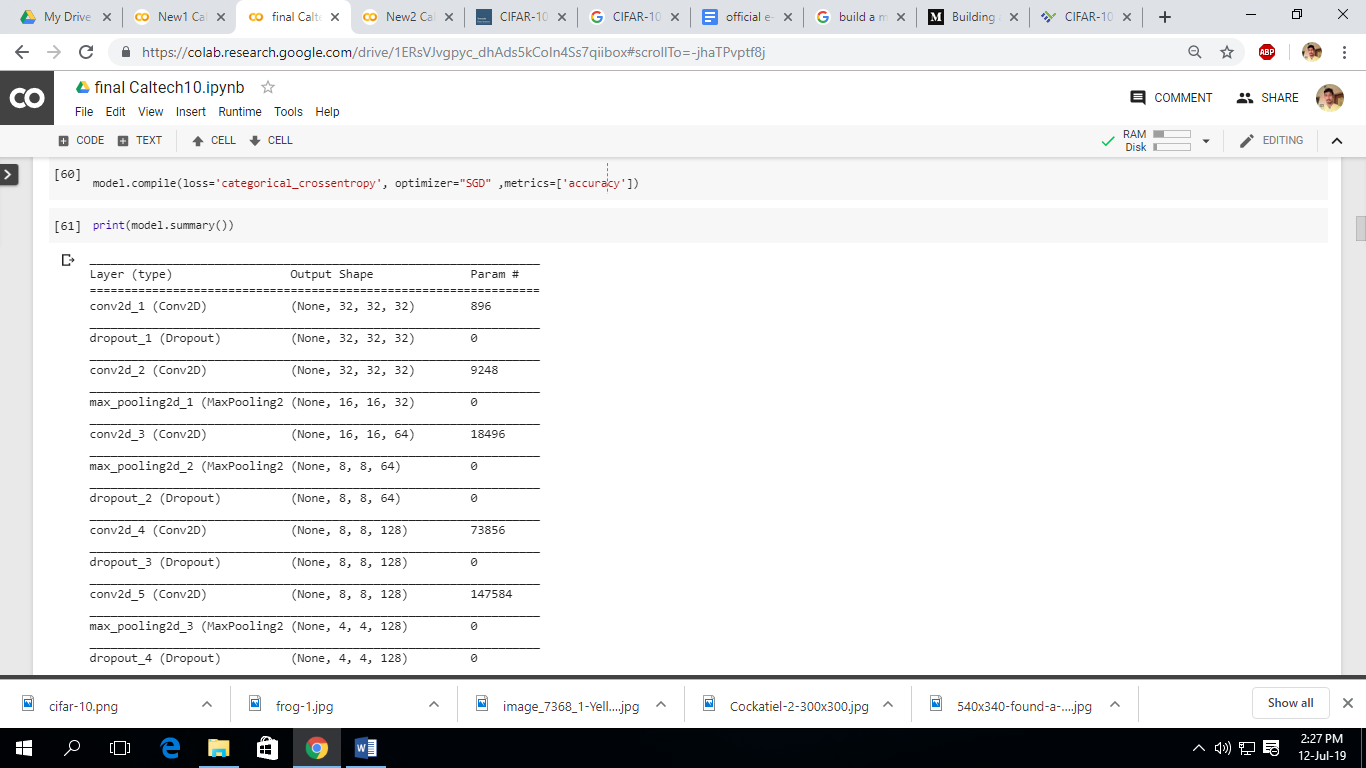
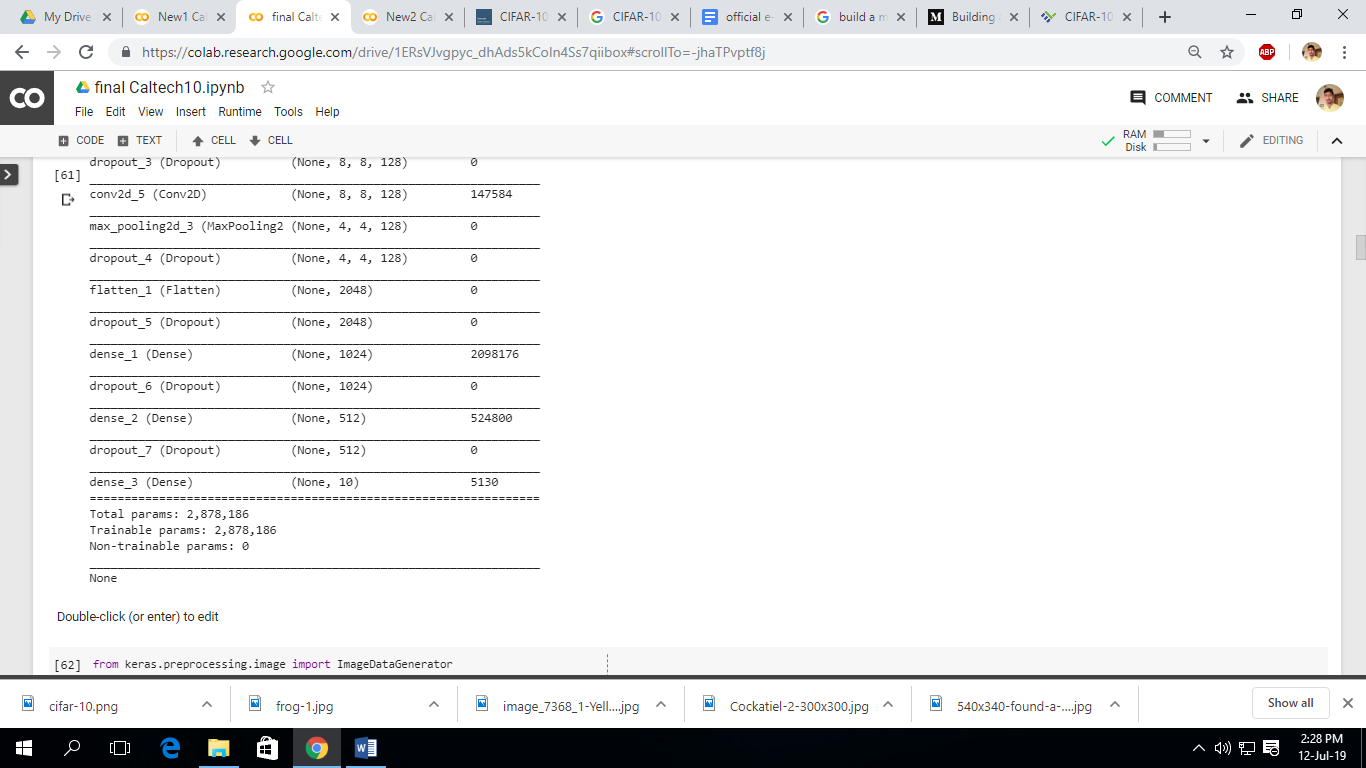
Now we will print the validation accuracy and loss

Here we can say that the accuracy using adam optimizer is 82.23%.

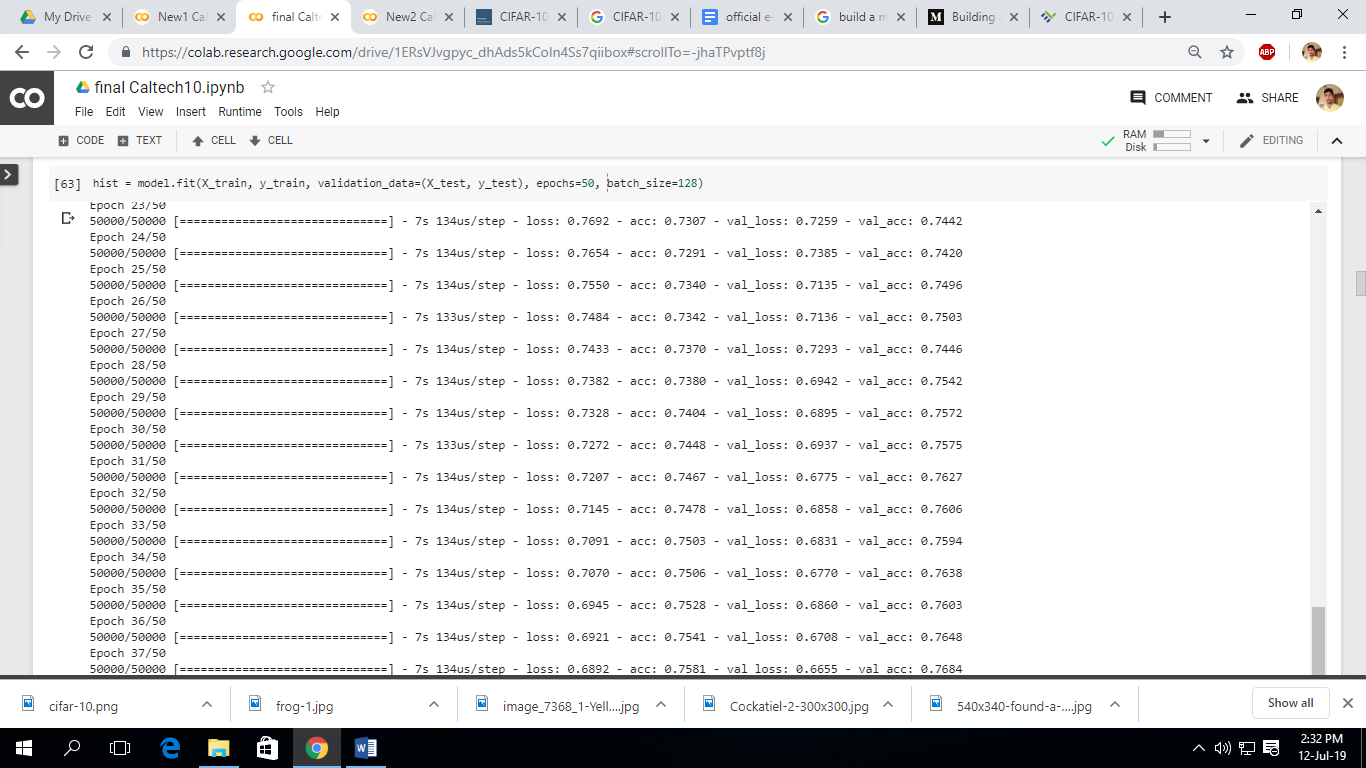
Now we plot the accuracy and loss with respect to epoch using the train and test data.

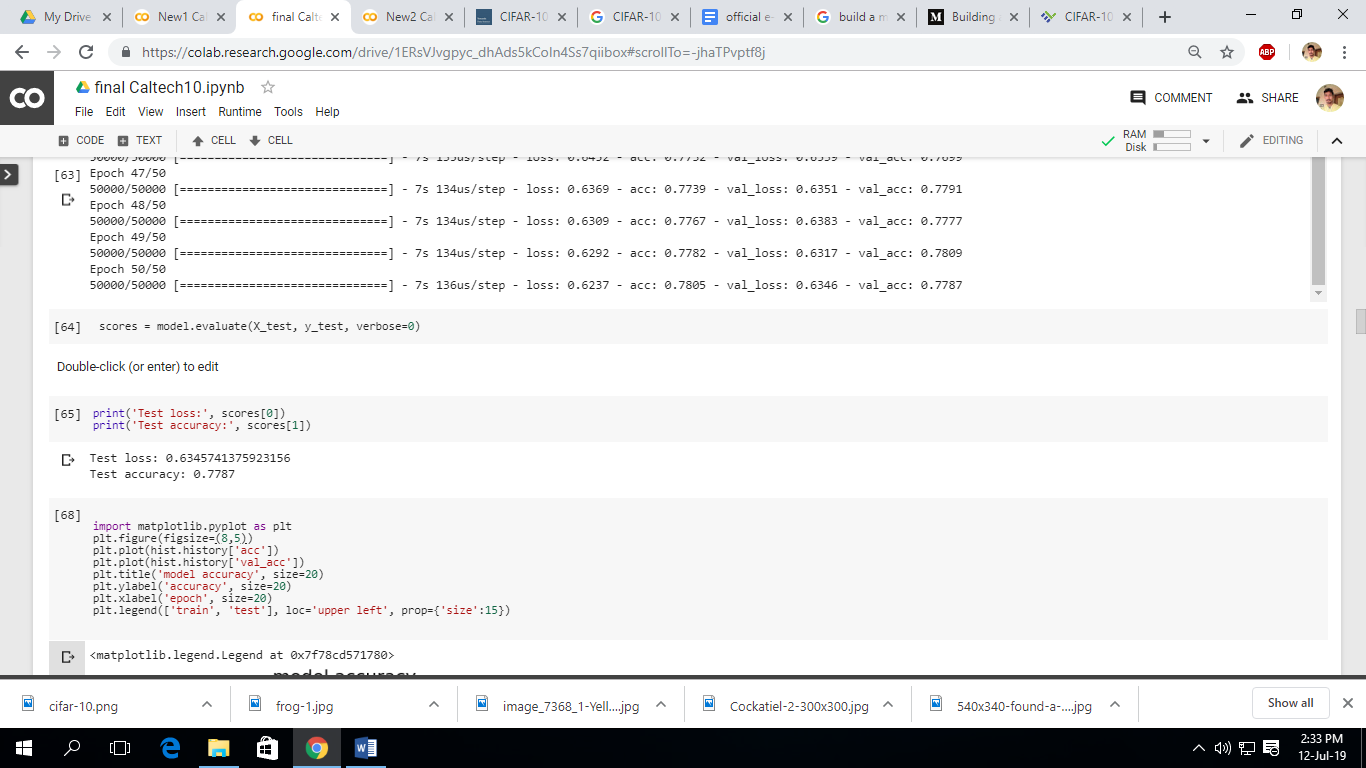


* Model using SGD

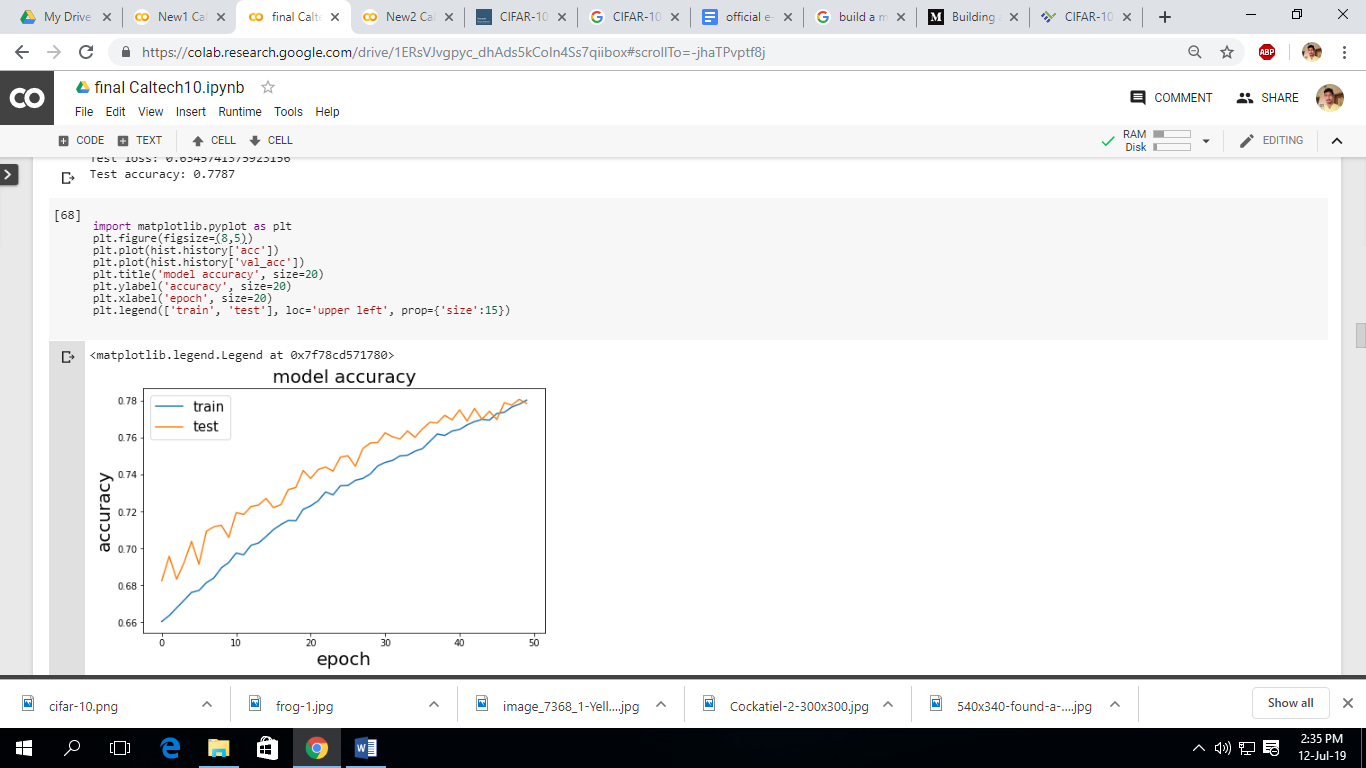
now, we compile and print the model summary

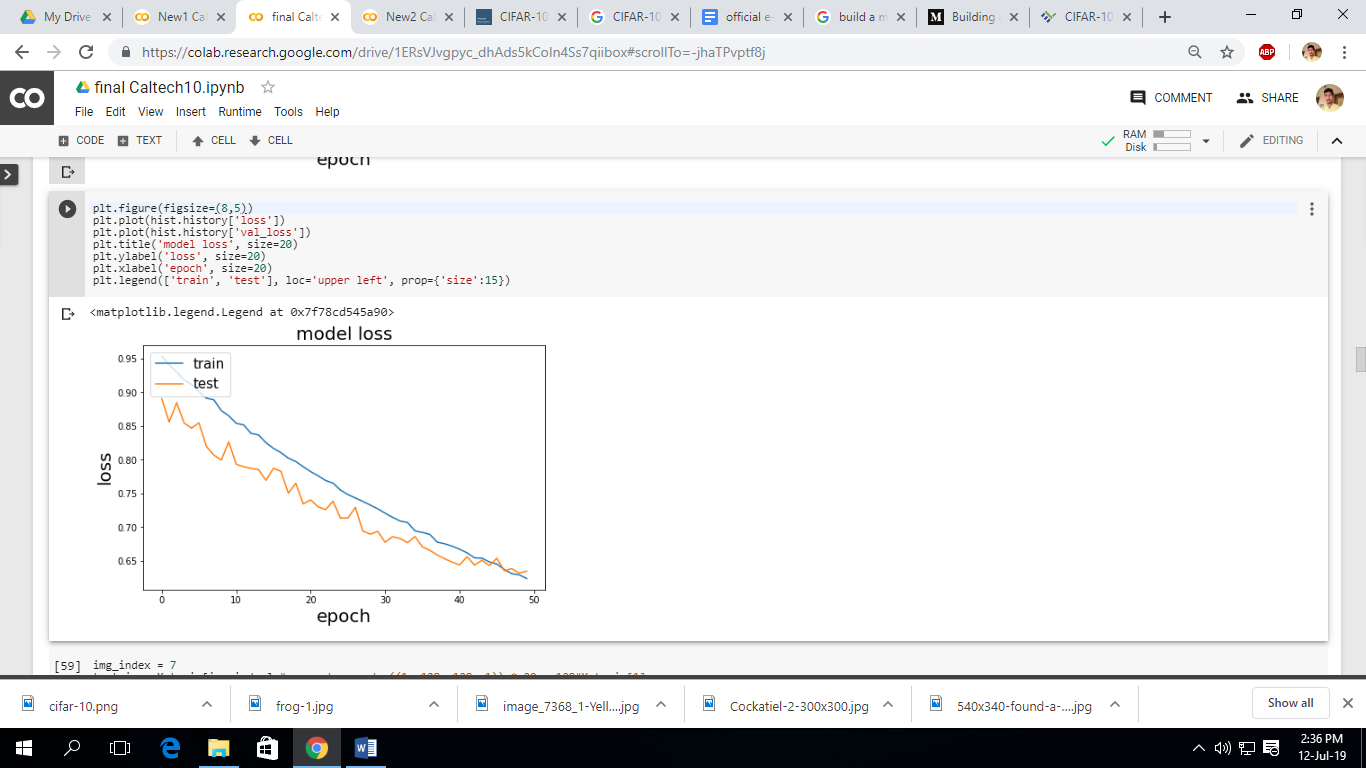
Now, we fit the model using validation and train data and we take 50 epochs and 128 batch size.



Now we will print the validation accuracy and loss

Here we can see the accuracy of model using SGD optimizer is 77.87%.

Now we plot the accuracy and loss with respect to epoch using the train and test data.



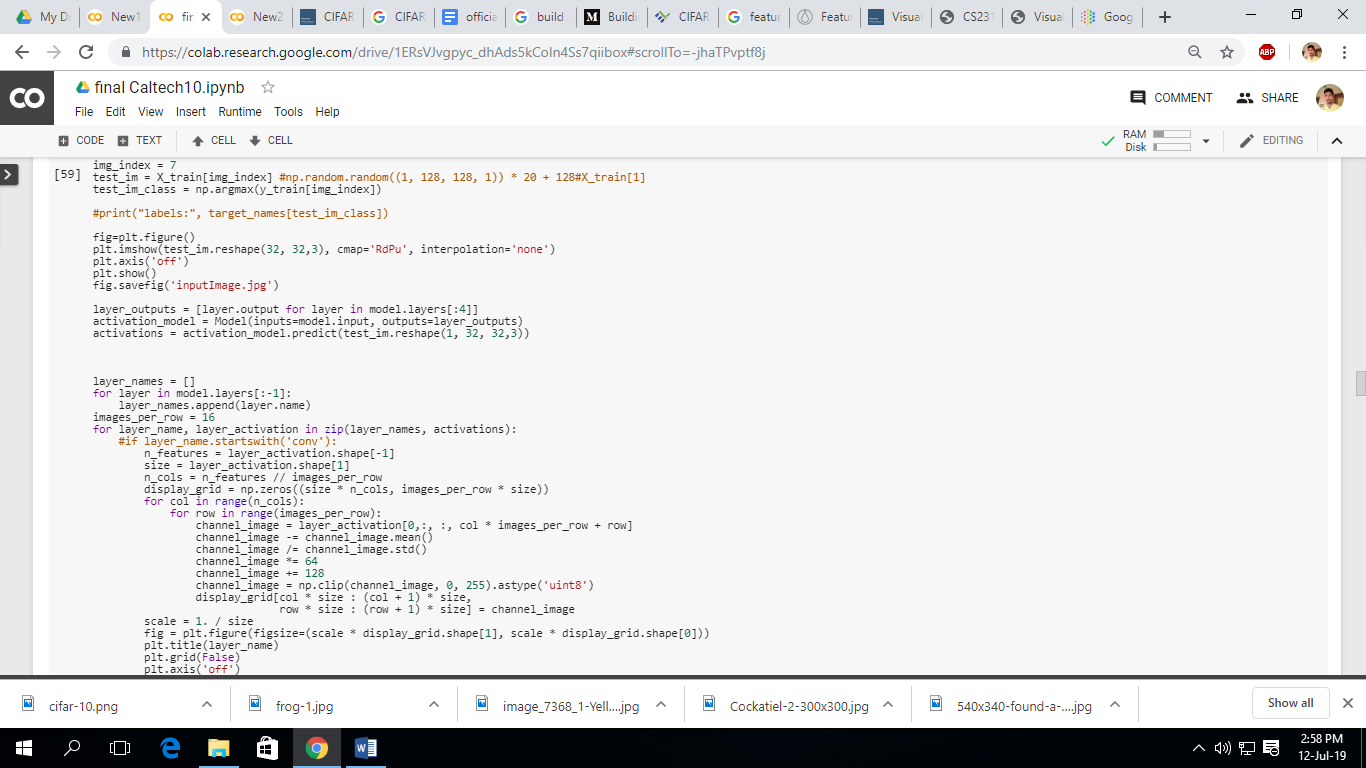
Now we can see that accuracy of model using Adam optimizer is better than SGD optimizer. So, here we use the Adam Optimizer for classification of data.

## Feature Visualization:

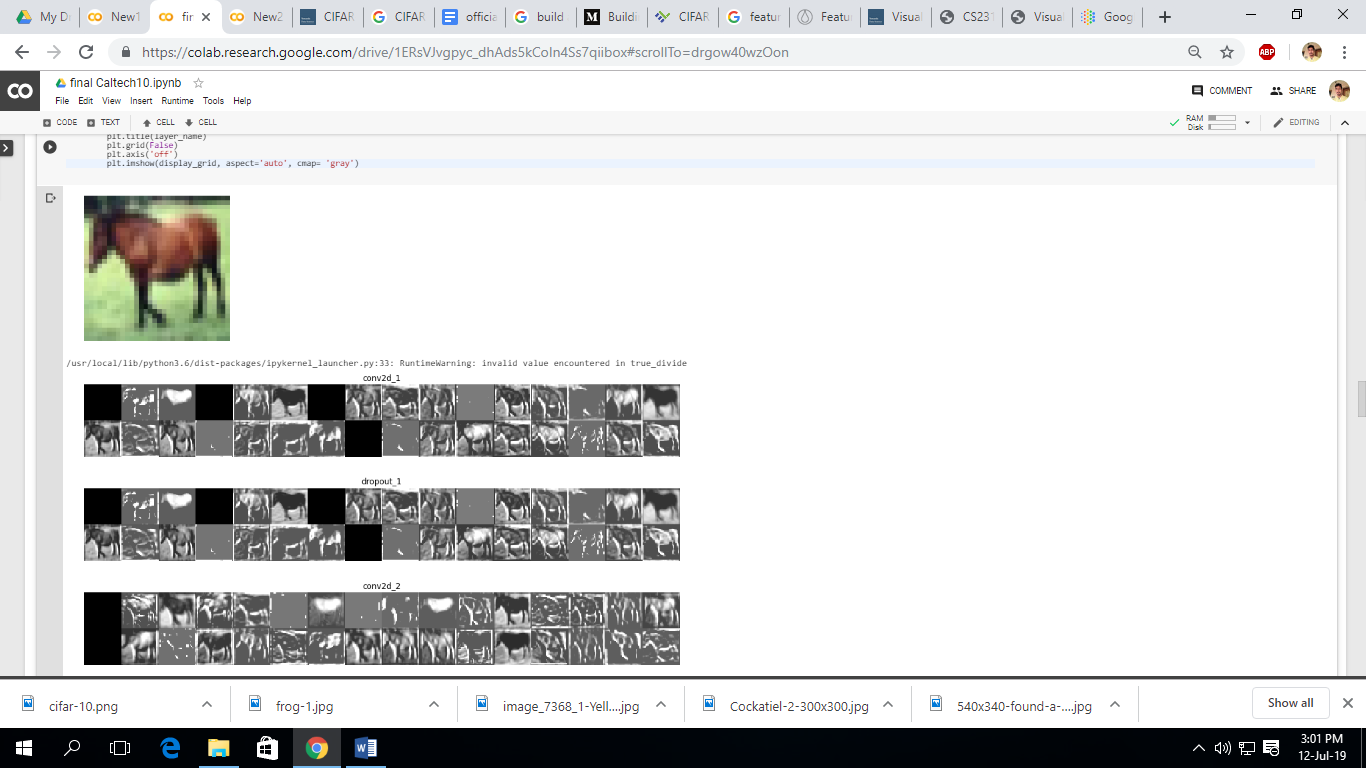
## Neural networks are, generally speaking, differentiable with respect to their inputs. If we want to find out what kind of input would cause a certain behavior — whether that’s an internal neuron firing or the final output behavior — we can use derivatives to iteratively tweak the input towards that goal.

## Specifically, we will try out various visualization techniques to understand

1. What is the output after each convolutional operation (Visualizing intermediate activations)
2. What features does each filter extract from the input image (Visualizing filters/kernels)
3. How to trace back output of each convolutional layer back to the input image
4. Visualizing areas of interest, in the input image, for each layer of convent



We only see the first 4 layers.

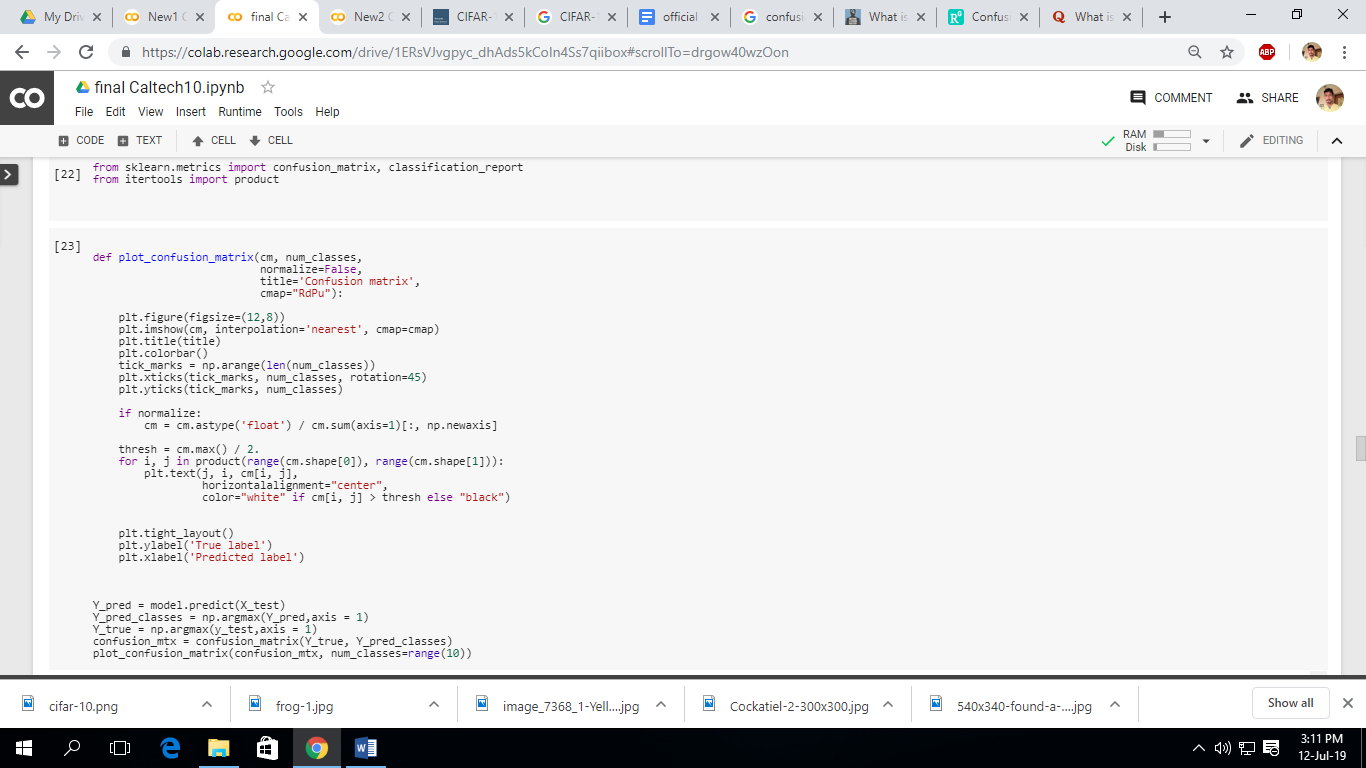


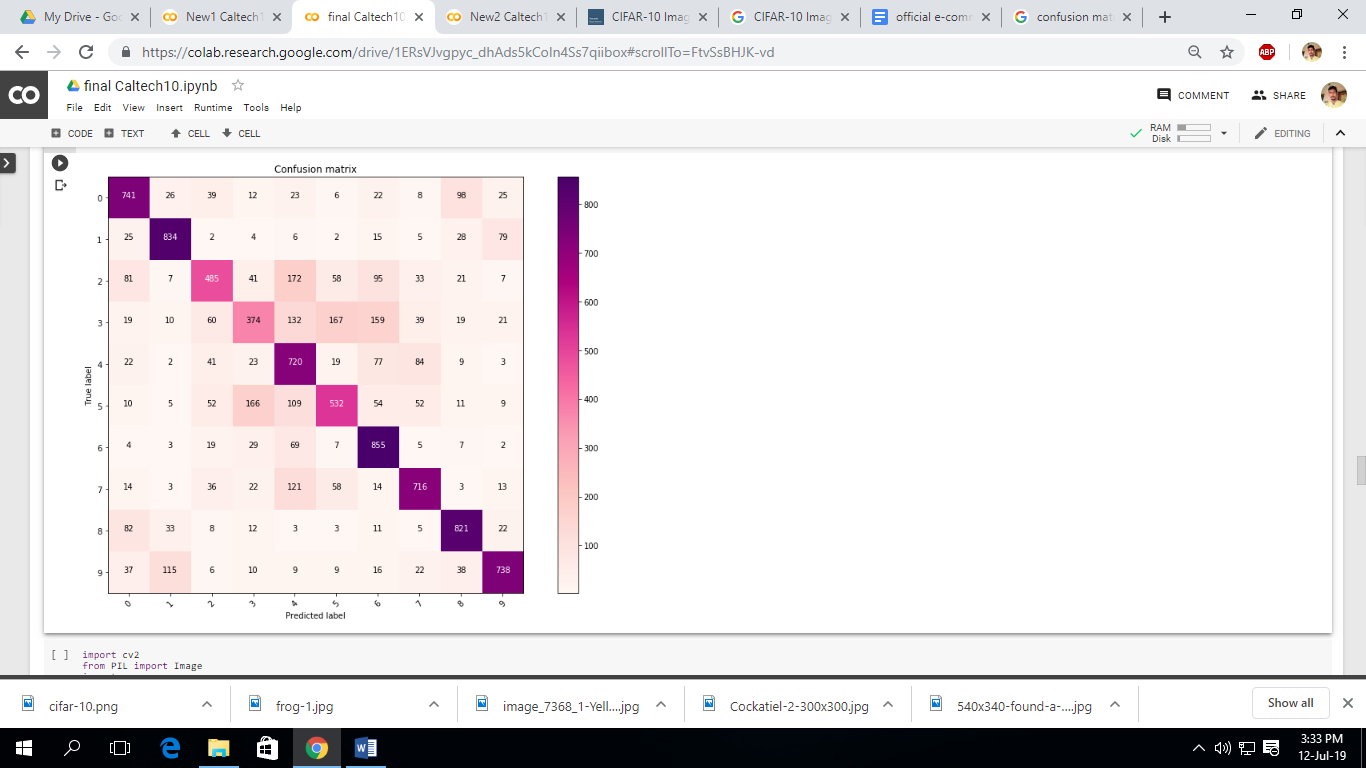
* **Model Performance**
* Confusion matrix

A confusion matrix is a technique for summarizing the performance of a classification algorithm.

Classification accuracy alone can be misleading if we have an unequal number of observations in each class or if we have more than ten classes in our dataset.

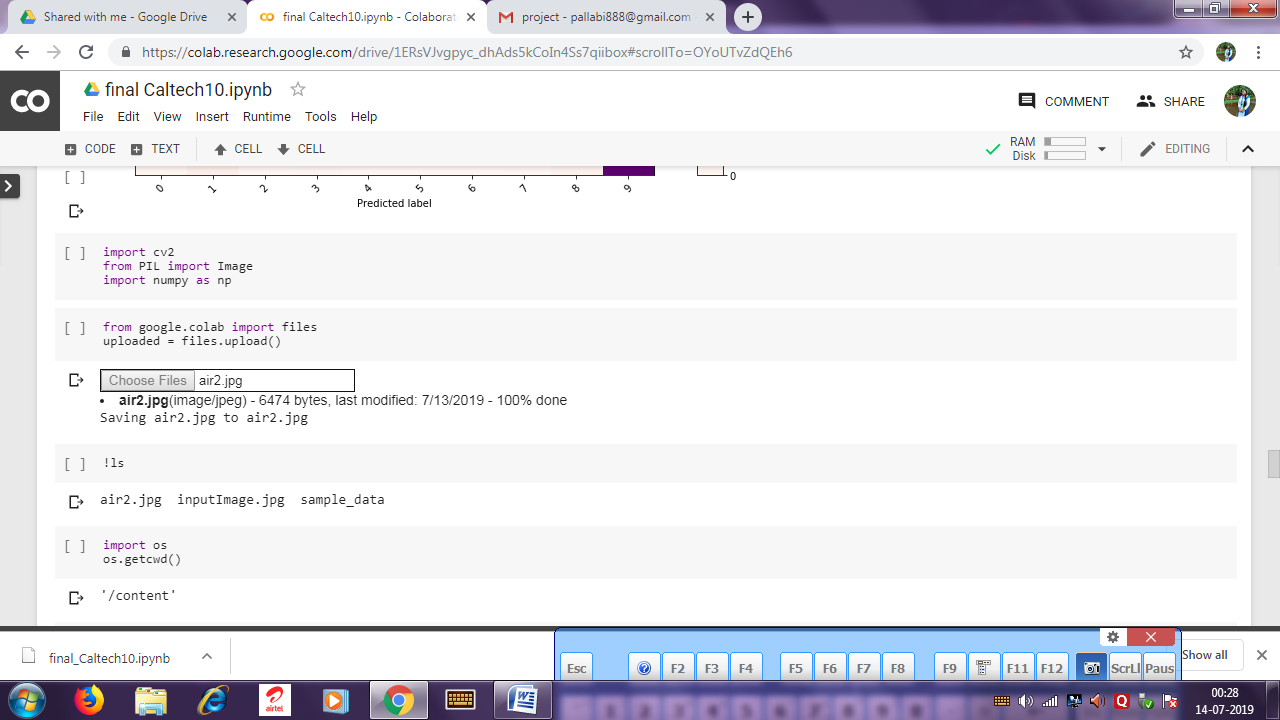
Calculating a confusion matrix can give us a better idea of what your classification model is getting right and what types of errors it is making.



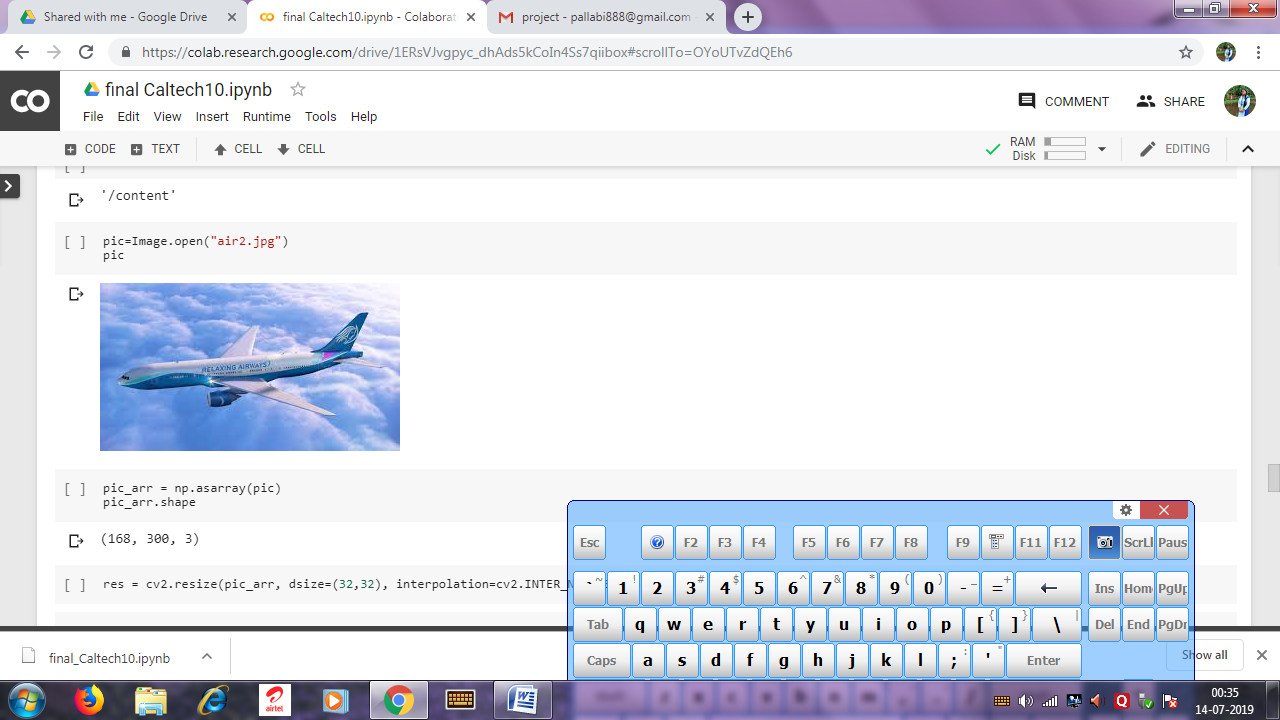


* **Test the model with our own Data**

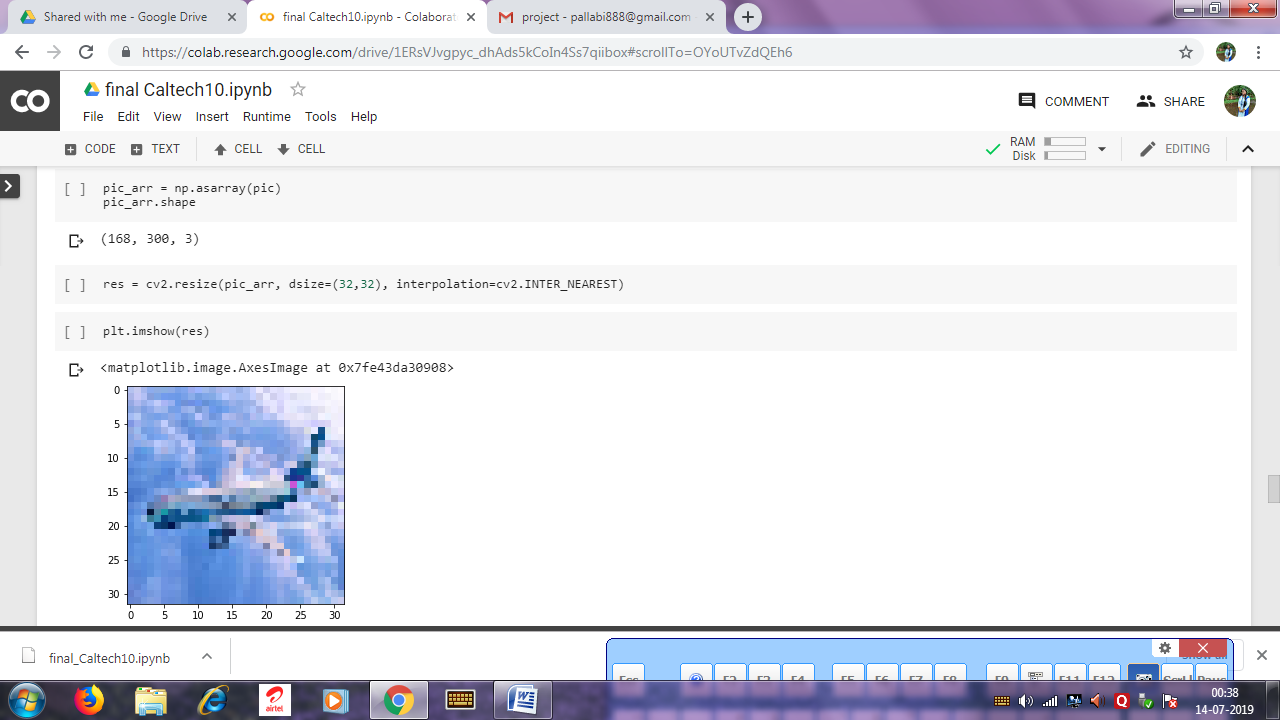
At first we have to load any image.



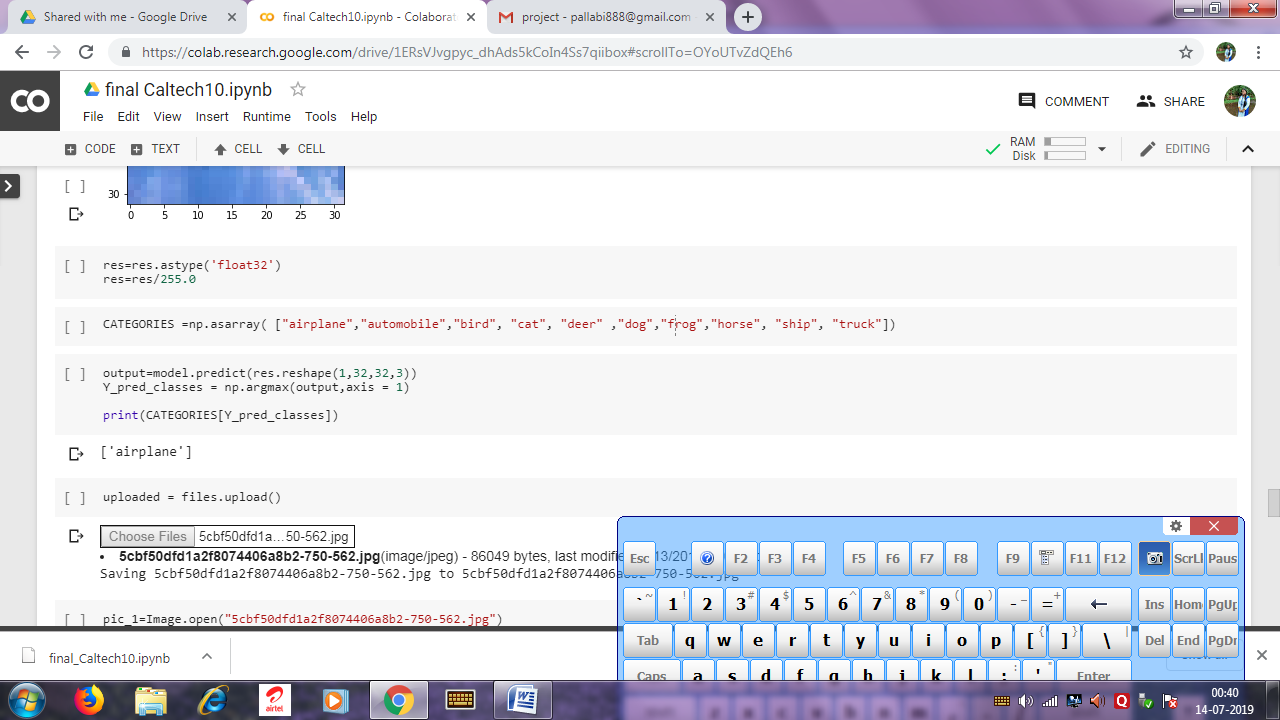
Next, we will open the image and look like following



Now, we will create the array and then reshape the image, the image will be look like,

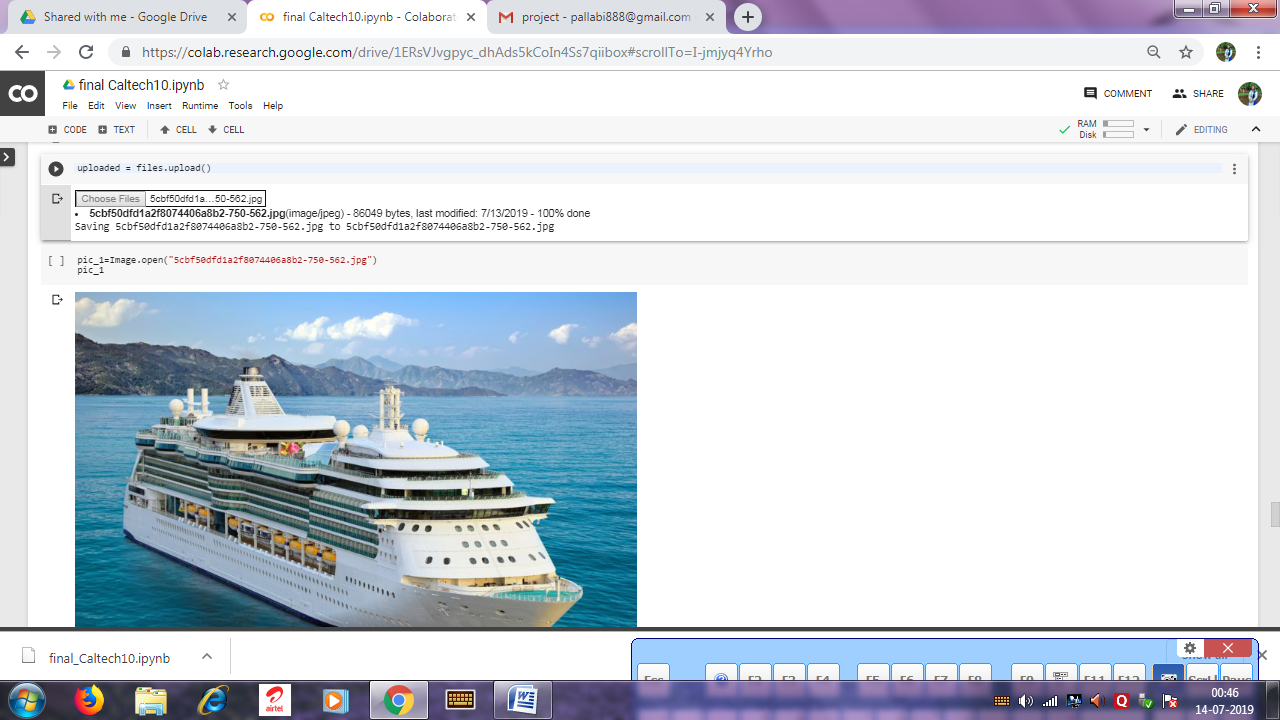


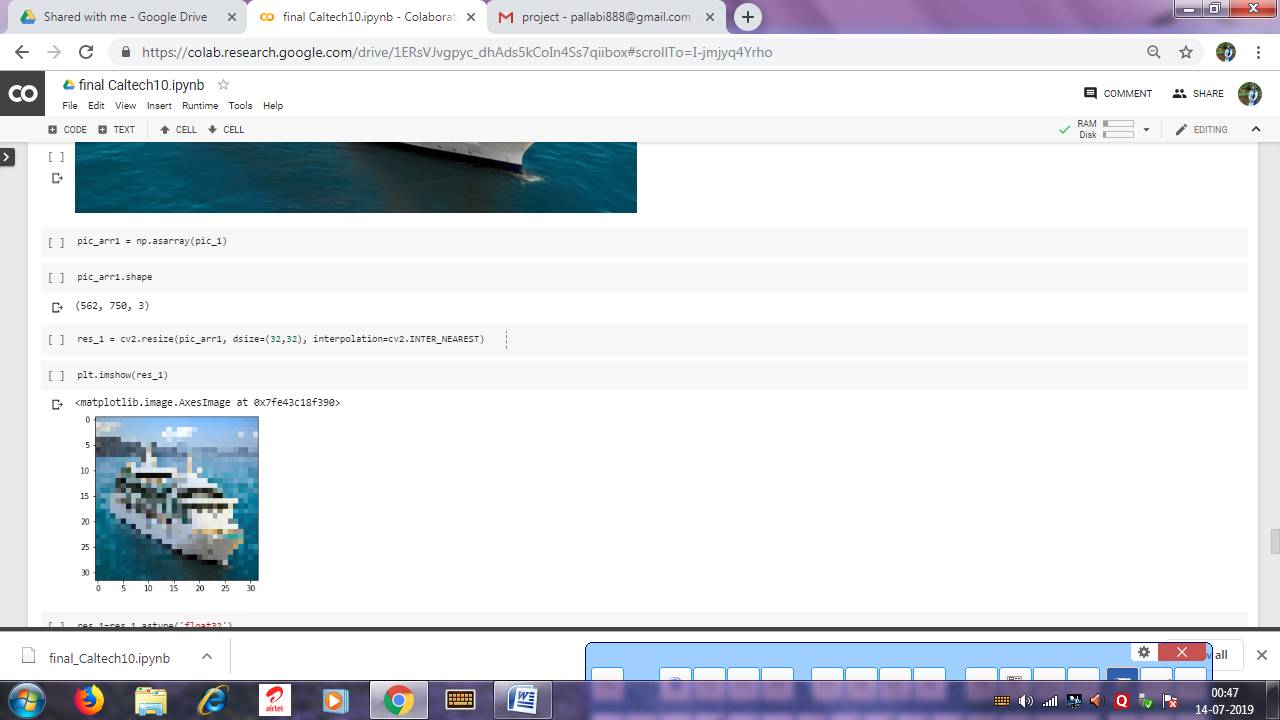
After that we run the image through our model and see the result.

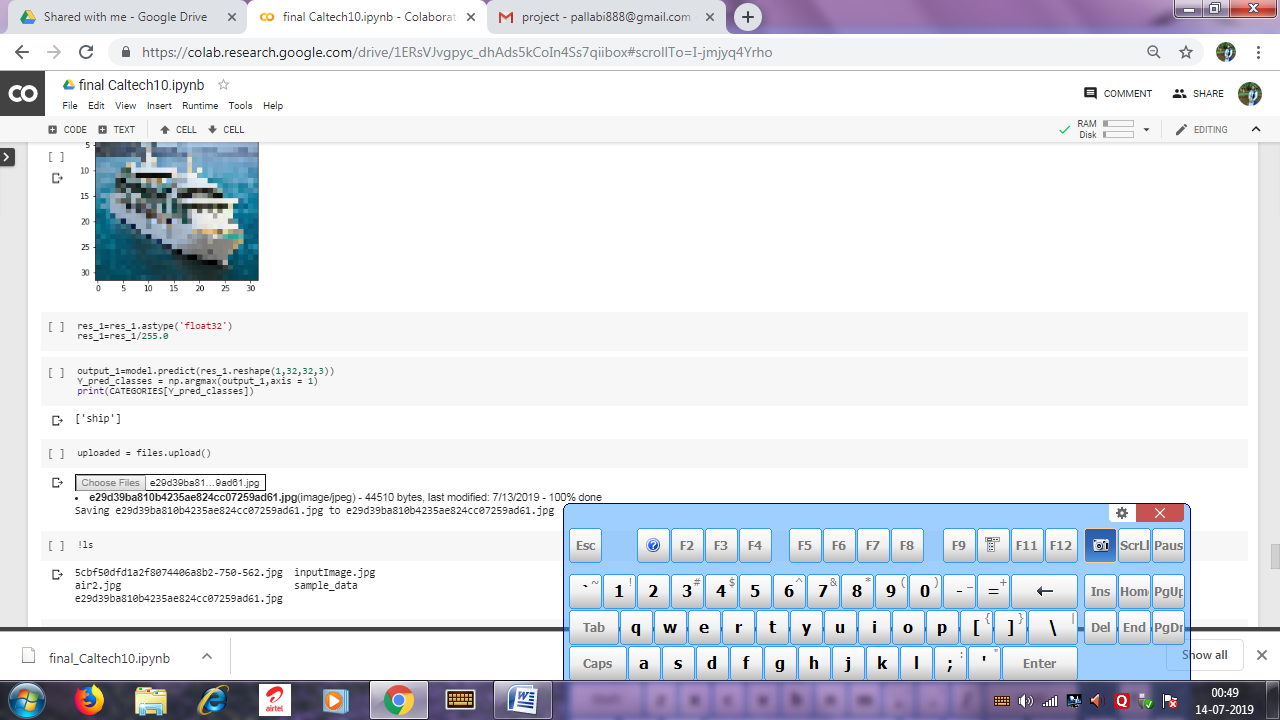


Here we can see that the output is ‘airplane’, that means our model is accurate.

Now we see another two examples of same process.

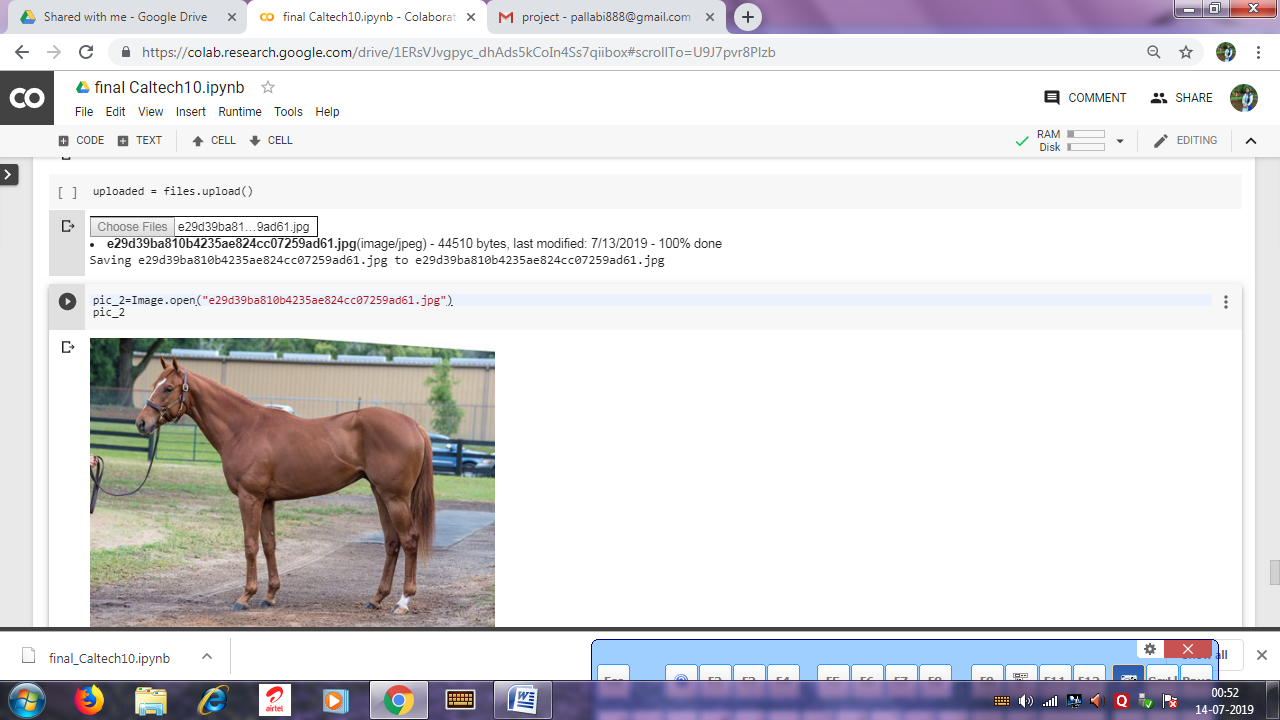




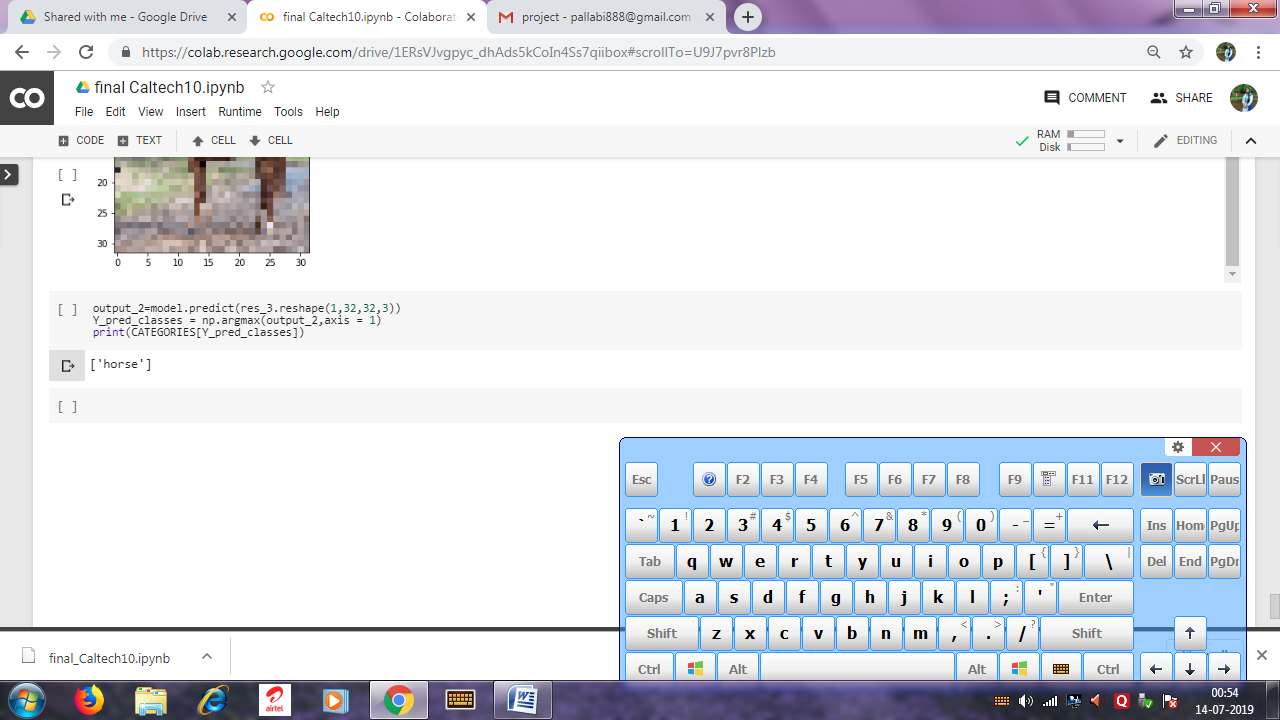


In this example we also got right output.

Now, another one,







So, the third one is also right.

* **Conclusion**

After seeing the entire example we can conclude that our model is accurate to classify each image of the dataset. Our model’s accuracy is 82%.