**1. INTRODUCTION**

Image recognition is used to perform a large number of machine-based visual tasks, such as labeling the content of images with [meta-tags](http://whatis.techtarget.com/definition/meta-description-tag), performing image content search and guiding autonomous robots, self-driving cars and accident avoidance systems.

While human and animal brains recognize objects with ease, computers have difficulty with the task. Software for image recognition requires [deep machine learning](http://searchbusinessanalytics.techtarget.com/definition/deep-learning). Performance is best on convolutional [neural net processors](http://searchnetworking.techtarget.com/definition/neural-network) as the specific task otherwise requires massive amounts of power for its compute-intensive nature. Image recognition [algorithms](http://whatis.techtarget.com/definition/algorithm) can function by use of comparative [3D models](http://whatis.techtarget.com/definition/3D-model), appearances from different angles using edge detection or by components. Image recognition algorithms are often trained on millions of pre-labeled pictures with guided computer learning. Thus the concepts of GoogLeNet in Keras will set the computational intelligence stage for some amazing architectures that could advance the fields of computer vision and image recognition.

Image recognition is one of the hallmark tasks of computer vision, allowing the definition of a context for object recognition.

The project carried is in the areas of Computer Vision with a deep convolutional neural network model in fields of Computer Science.

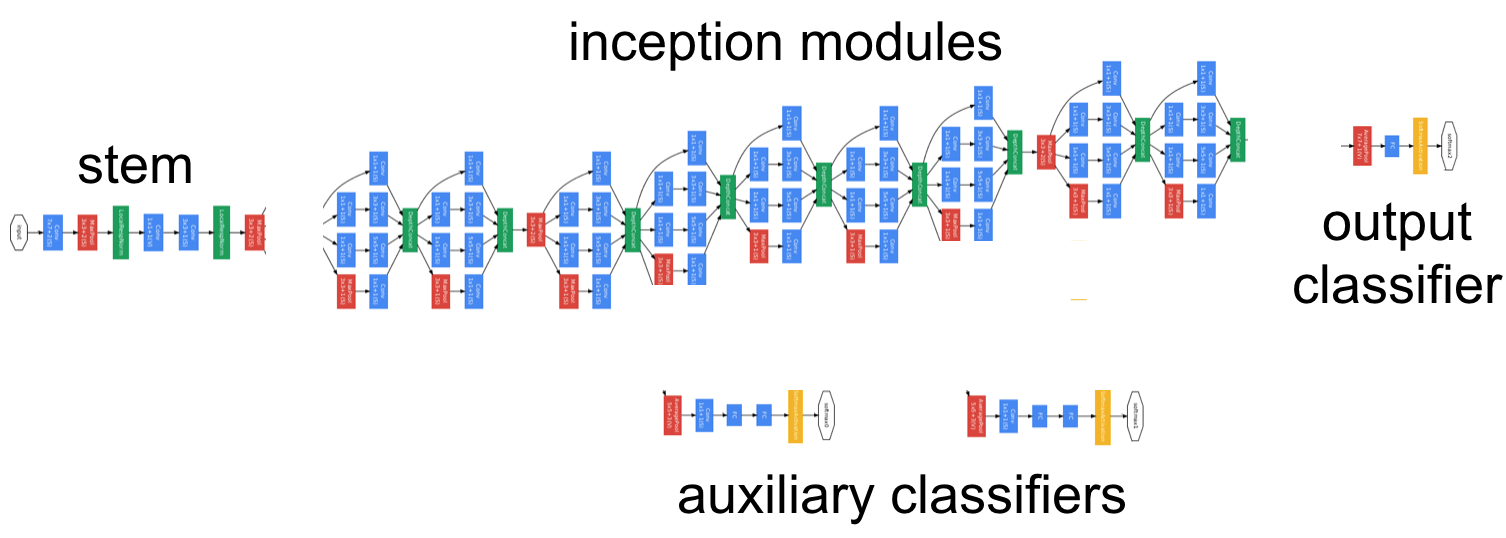
**1.1 Objective**

The main objective of this project is to simulate the existing research on GoogLeNet. In general, CNNs are always sequential so an Inception (a deep network model) is used for creative structuring of layers. To simulate using Keras running on top of Theano in experimenting to uncover some new knowledge in the latest research area: “Going deeper with Convolutions - GoogLeNet” for image recognition. In order to lead to improved performance and computationally efficiency in recognition of images.

The concepts of GoogLeNet in Keras will set the computational intelligence stage for some amazing architectures that could advance the fields of computer vision and image recognition.

**1.2 Methodology**

**GoogLeNet in Keras**

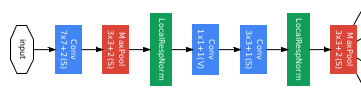
****

**Figure 1.1: GoogLeNet in Keras**

The behemoth that sits is GoogLeNet. As part of an ensemble of other similar models trained by the researchers at Google, GoogLeNet achieved a top-5 error rate of 6.67% on the 2014 ImageNet classification challenge. What that means is this : if you have an image of an object that is contained in the 1,000 object classes of the ImageNet dataset (all sorts of animals, household objects, vehicles, etc.), 93.33% of the time the correct object class will be contained in the GoogLeNet ensemble's top five predictions. Considering that ImageNet consists of many fine-grained object categories and that some images contain multiple object categories, this is an incredible feat, nearly on par with human performance. While at first glance the model may appear incredibly complex, upon closer inspection, the overall structure of the model can be broken down into a few basic sections: the stem, the inception modules, the auxiliary classifiers, and finally the output classifier.

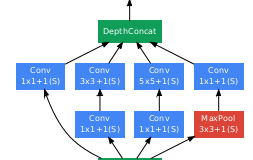
The basic building block of GoogLeNet, the inception module, is a set of convolutions and poolings at different scales, each done in parallel, then concatenated together. Along the way, 1×1 convolutions are used to reduce the dimensionality of inputs to convolutions with larger filter sizes. This approach results in a high performing model with drastically fewer parameters. GoogLeNet, in fact, has a factor of 12 times fewer parameters than AlexNet.

The stem stands in contrast to the rest of the GoogLeNet architecture, which is primarily made up of what are referred to as 'inception' modules.

****

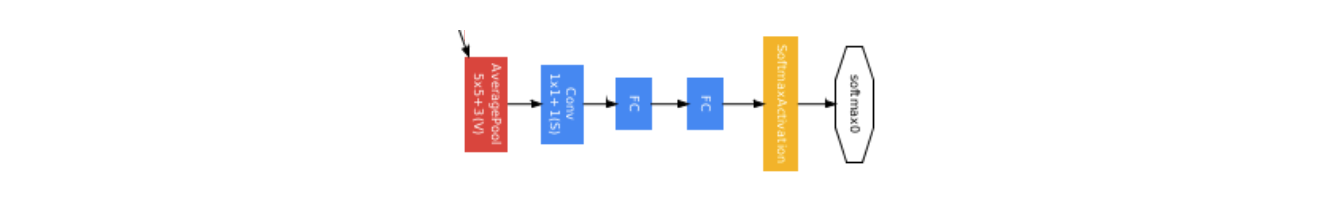
**Figure 1.2 : Stem Architecture**

GoogLeNet contains nine of these modules, sequentially stacked, with two max pooling layers along the way to reduce the spatial dimensions. Due to the depth of this architecture, the authors added two auxiliary classifiers branching from the main network structure. The purpose of these classifiers is to amplify the gradient signal back through the network, attempting to improve the earlier representations of the data.

****

**Figure 1.3: Depth concatenation Architecture**

Finally, we get to the output classifier, which performs an average pooling operation followed by a softmax activation on a fully connected layer.

****

**Figure 1.4 : Output Classifier**

In total, the network uses the standard operations: convolution, pooling, normalization, and fully-connected layers.

**1.3 Organisation of Project**

As the project is a simulation on images. It involves tasks like recognition and extraction of features to define the image’s class.

We divided our project into 2 major modules.

* Firstly, the task is building the layers of Inception module to focus in depth of an image.
* Secondly, recognition of image using Theano as backend for Keras API model which provides rich library of image processing in-built functions.

**2. Theoretical Analysis of proposed project**

**2.1 System Specifications and features**

HARDWARE :

Operating System : Linux/Ubuntu

Processor : Intel core i5

CPU speed : 2.2 GHZ

Memory (RAM) : 8GB

SOFTWARE :

Programming Language : Python 2.7

Neural network API : Keras 2.0

Backend-tool framework : Theano 1.0

Architecture : Inception v1 - GoogLeNet

Classes : Electronic goods and Vehicular objects

**2.1.1 Technology Description**

**Python**

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently whereas other languages use punctuation, and it has fewer syntactic constructions than other languages.

* **Python is Interpreted** − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
* **Python is Interactive** − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.

Python also acknowledges that speed of development is important. Readable and terse code is part of this, and so is access to powerful constructs that avoid tedious repetition of code. Maintainability also ties into this may be an all but useless metric, but it does say something about how much code you have to scan, read and/or understand to troubleshoot problems or tweak behaviors. This speed of development, the ease with which a programmer of other languages can pick up basic Python skills, and the huge standard library is key to another area where Python excels: **Tool Making**. Any project of size will have tasks to automate, and automating them in. All of those tools have been quick to implement, saved a lot of time, and several of them have later been patched and updated by people *with no Python background* - without breaking.

**Step 1: Installation of Python**

The first step in Python's installation is running a configure script which snoops around your system, looking for packages that it needs to build various capabilities and extensions with. Having these packages installed before running configure makes sure it finds them.

Here are some packages that have to be installed to have various aspects of Python functioning:

> sudo apt-get install libreadline-dev

> sudo apt-get install libsqlite3-dev

> sudo apt-get install libbz2-dev

> sudo apt-get install libssl-dev

**Step 2: Download and build Python**

Go to <http://www.python.org/>. In the "Quick links" section on the left-hand side of the page, "Source distribution" is a direct link to the tarball. Download it. Unzip the tarball, and from the root of the created directory (which will be called Python-2.7.2 or something similar, depending on the version):

./configure

make -j

You can now check that Python was correctly built by executing ./python and falling into its interactive terminal. If you want, you can also execute the Python test-suite with make test, though it may take a long time to run (~10 minutes on a relatively fast machine).

**Step 3: Install**

In the same directory, run:

sudo make install

This installs Python into /usr/local/bin. Depending on the configuration of your system, you may want to add symlinks to the newly created /usr/local/bin/python2.7 in /usr/bin/ as well.

That's it, you now have Python 2.7 installed.

**Step 4: Install some essential first modules**

Python has a powerful packaging & installation machinery for its modules, but it doesn't come pre-installed with Python itself.

So it's a good idea to install [setuptools](http://pypi.python.org/pypi/setuptools) (or [distribute](http://pypi.python.org/pypi/distribute)), followed by [pip](http://pypi.python.org/pypi/pip).

From now, pip can be used to install other Python modules very conveniently. For example, all you need to have the IPython shell installed is:

sudo pip install ipython

**SciPy**

SciPy is package of tools for science and engineering for Python. It includes modules for statistics, optimization, integration, linear algebra, Fourier transforms, signal and image processing, ODE solvers, and more. SpiceyPy is currently supported on Mac, Linux, and Windows systems.

The basic data structure used by SciPy is a multidimensional array provided by the [NumPy](https://en.wikipedia.org/wiki/NumPy) module. NumPy provides some functions for linear algebra, Fourier transforms and random number generation, but not with the generality of the equivalent functions in SciPy. NumPy can also be used as an efficient multi-dimensional container of data with arbitrary data-types. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases. Older versions of SciPy used Numeric as an array type, which is now deprecated in favor of  the newer NumPy array code attempt to easy\_install indicates a problem with their [listing](http://pypi.python.org/pypi/scipy/0.7.0) in the [Python Package Index](http://pypi.python.org/pypi), which pip searches.

*Installation of SciPy :*

*> sudo pip install numpy*

*> sudo pip install scipy*

**Keras**

Keras is a high-level neural networks API, written in Python and capable of running on top of [TensorFlow](https://github.com/tensorflow/tensorflow), [CNTK](https://github.com/Microsoft/cntk), or [Theano](https://github.com/Theano/Theano). It was developed with a focus on enabling fast experimentation.

Keras models are defined by two files: a json file containing the model architecture and an hdf5 file containing the model's weights.

The Keras functional API is the way to go for defining complex models, such as multi-output models, directed acyclic graphs, or models with shared layers.

**Installation of keras**

The good news is that if you used Anaconda, then you'll already have a nice package management system called **pip** installed.

You can confirm you have it installed by typing  $ pip in your command line. It should output a list of commands and options. If you don't have pip, you need to first install pip

Once you have pip, installing Keras is easy as pie:

|  |  |
| --- | --- |
|  | $ pip install keras  To see whether keras is installed or not |

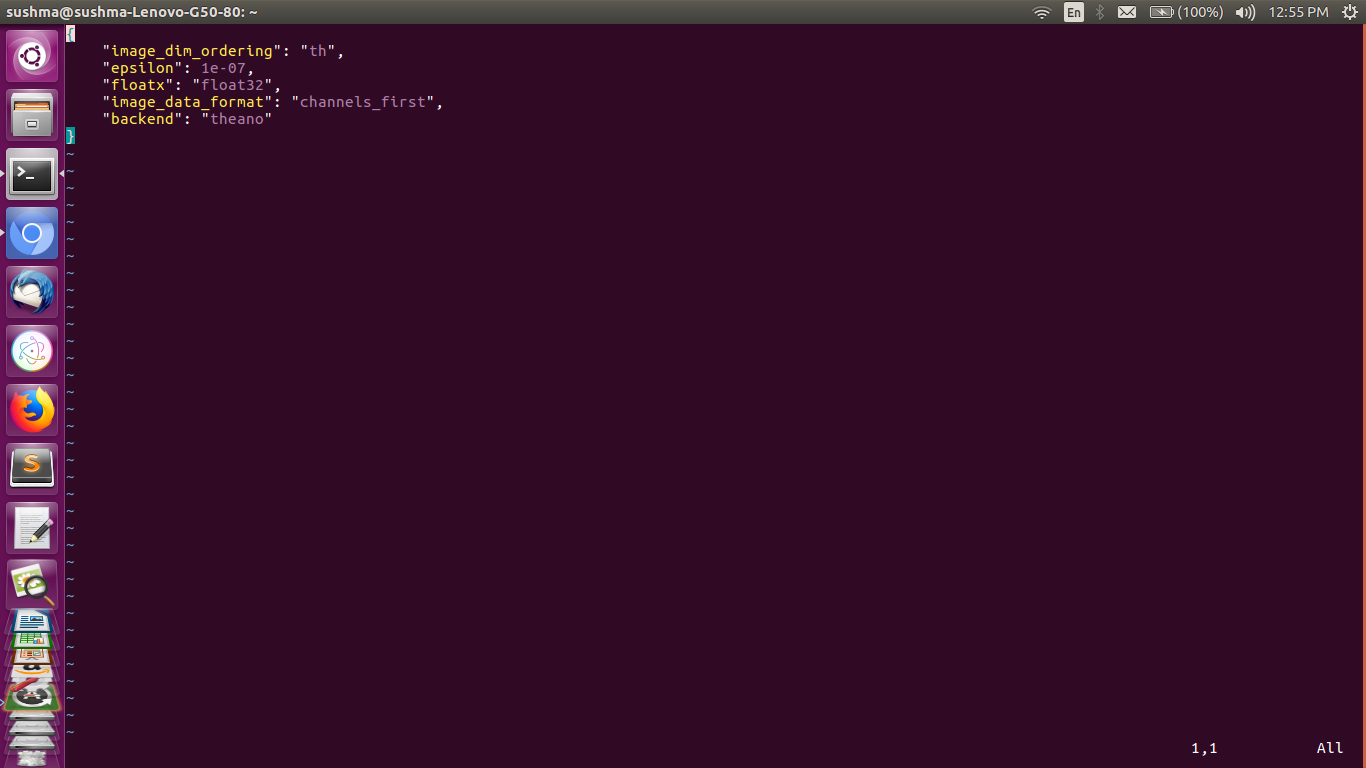
$ python -c "import keras; print keras.\_\_version\_\_"

Using Theano backend.

1.0.4

To update keras version

$ pip install --upgrade keras



**Figure 2.1: Configuration of Theano as backend in Keras**

$ python -c "import keras; print keras.\_\_version\_\_"

Using Theano backend.

1.1.1

**Theano**

Theano is a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently. Theano features:

* **tight integration with NumPy** – Use *numpy.ndarray* in Theano-compiled functions.
* **transparent use of a GPU** – Perform data-intensive computations much faster than on a CPU.
* **efficient symbolic differentiation** – Theano does your derivatives for functions with one or many inputs.
* **speed and stability optimizations** – Get the right answer for log(1+x) even when x is really tiny.
* **dynamic C code generation** – Evaluate expressions faster.
* **extensive unit-testing and self-verification** – Detect and diagnose

Theano combines aspects of a computer algebra system (CAS) with aspects of an optimizing compiler. It can also generate customized C code for many mathematical operations. This combination of CAS with optimizing compilation is particularly useful for tasks in which complicated mathematical expressions are evaluated repeatedly and evaluation speed is critical. For situations where many different expressions are each evaluated once Theano can minimize the amount of compilation/analysis overhead, but still provide symbolic features such as automatic differentiation

Theano is not a programming language in the normal sense because you write a program in Python that builds expressions for Theano. Still it is like a programming language in the sense that you have to

* declare variables (a,b) and give their types
* build expressions for how to put those variables together
* compile expression graphs to functions in order to use them for computation.

It is good to think of theano.function as the interface to a compiler which builds a callable object from a purely symbolic graph. One of Theano’s most important features is that theano.function can optimize a graph and even compile some or all of it into native machine instructions.

Theano was initially created by a machine learning research group from University of Montreal and is currently actively developed by Theano community. The essential part of Theano is operations on multidimensional arrays that can be easily parallelized. High parallelization is especially important for more efficient utilization of GPUs that typically have thousands of simple computing units. This is addressed with a computational graph which is a core concept of computing in Theano. The graph represents all computations in the code of the model where graph nodes are either variable type (tensor) or operations (e.g. dot product or sigmoid function) on tensors(multidimensional arrays of data). During compilation Theano optimizes the graph and generates high performance C code for each operation node. If your hardware has CUDA support Theano can generate CUDA by a simple change in configuration. After the model is compiled, Python code can use it for efficient computations, passing the data to variable nodes in the computation graph and receiving results. On the Figure 1 you can see a computation graph for a simple model of Neural Network that learns logical XOR operation.

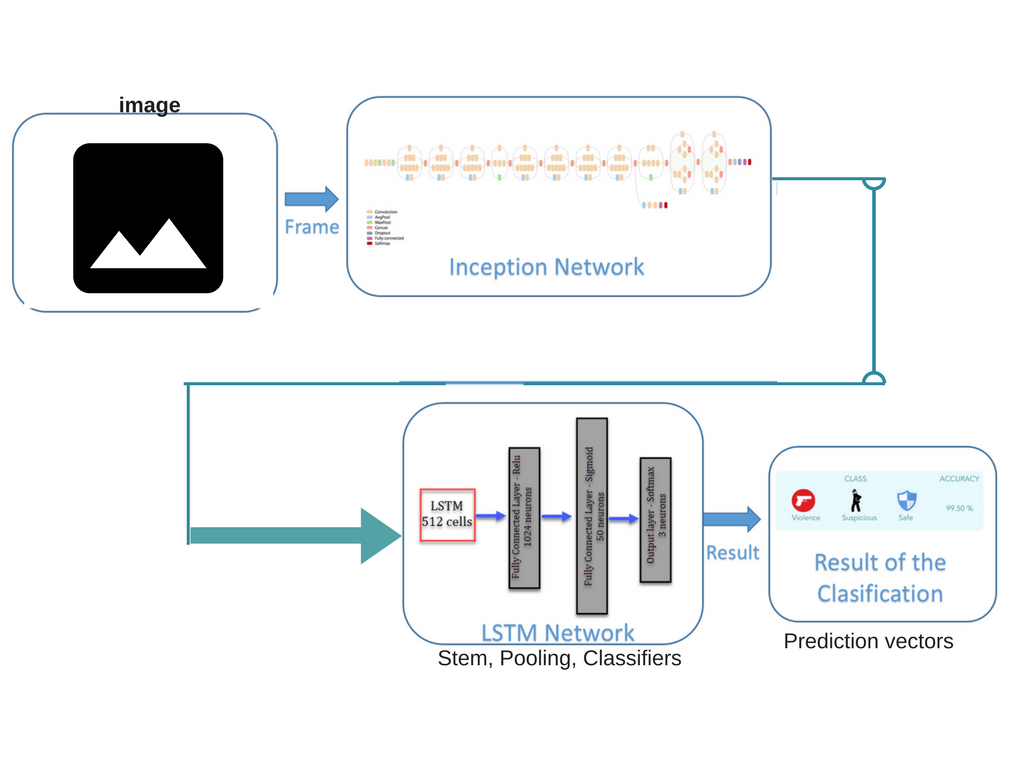
**2.2 Architecture Diagram**

To conduct simulations and to have exposure on new concept called “Inception v1” network model and technology in the area of research for Computer Vision.

The main tasks to be implemented are

* To recognize an image’s class using layers of Inception network model.
* To assess the quality of pre-trained image database : GoogLeNet.

The architecture of project typically depicts the way the simulation happens. First the image is inputted to the model, immediately the image dimensions are reduced to a fixed dimension to maintain uniformity, given any image type. Next the image is passed down to layers of convolutional networks which act as network inside a network i.e, Inception model. Filters will be applied on the input image while passing through these convolutional layers which calculates weights. Next, all these weights are pooled up through Long Short-Term Memory (LSTM) Network layer. Finally the weight as a single value is outputted to GoogLeNet image labels. This single-weight value is matched with its respective class label of 1000 pre-trained class labels in GoogLeNet. Then, the class name of the input image is given as output. Finally, the image is recognized and it’s deep features are predicted.

****

**Figure 2.2 : Project Architecture**

**2.3 Software Requirements Specifications**

**Table 1 : Software Requirement Specifications - I**

|  |  |
| --- | --- |
| **Priority** | Essential |
| **Effort** | 3 weeks |
| **Description** | * The Initial segment : STEM is extracted from the existing GoogLeNet architecture. * Then the further model is built i.e, the Convolutional Neural Networks each of which has Max Pooling, Activations on fully connected layers. * Finally, the depth concatenation of all these layers is done. |

**Table 2 : Software Requirement Specifications - II**

|  |  |
| --- | --- |
| **Priority** | Essential |
| **Effort** | 4 weeks |
| **Description** | * 1 × 1, 3 × 3, and 5 × 5 convolutions * The outputs of these different pathways are concatenated * The next inception module will then take its input from this concatenated layer and so on. |

**3. Low Level Design**

**3.1 Image Recognition**

In this module, we aim for recognition of image’s class label using convolutional network.

**3.1.1 Construction of Convolutional Networks**

Convolutional Neural Networks are very similar to ordinary Neural Networks from they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer and all the tips/tricks we developed for learning regular Neural Networks still apply.

ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.

**Architecture**

Neural Networks receive an input (a single vector), and transform it through a series of *hidden layers*. Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer, and where neurons in a single layer function completely independently and do not share any connections. The last fully-connected layer is called the “output layer” and in classification settings it represents the class scores.

Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth. (Note that the word *depth* here refers to the third dimension of an activation volume, not to the depth of a full Neural Network, which can refer to the total number of layers in a network.) For example, the input images in CIFAR-10 are an input volume of activations, and the volume has dimensions 32x32x3 (width, height, depth respectively). As we will soon see, the neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner. Moreover, the final output layer would for CIFAR-10 have dimensions 1x1x10, because by the end of the ConvNet architecture we will reduce the full image into a single vector of class scores, arranged along the depth dimension. Here is a visualization:

## http://cs231n.github.io/assets/nn1/neural_net2.jpeg

## Figure 3.1 : ConvNet internal layer architecture

## http://cs231n.github.io/assets/cnn/cnn.jpeg

## Figure 3.2 : Convolution network flow

## Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

## Layers used to build ConvNets

A simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We use three main types of layers to build ConvNet architectures: **Convolutional Layer**, **Pooling Layer**, and **Fully-Connected Layer** (exactly as seen in regular Neural Networks). We will stack these layers to form a full ConvNet **architecture**.

A simple ConvNet for CIFAR-10 classification could have the architecture [INPUT - CONV - RELU - POOL - FC]. In more detail:

* INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R, G, B.
* CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x12] if we decided to use 12 filters.
* RELU layer will apply an element wise activation function, such as the max(0,x)max(0,x) thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]).
* POOL layer will perform a down sampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].
* FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

In this way, ConvNets transform the original image layer by layer from the original pixel values to the final class scores. Note that some layers contain parameters and other don’t. In particular, the CONV/FC layers perform transformations that are a function of not only the activations in the input volume, but also of the parameters (the weights and biases of the neurons). On the other hand, the RELU/POOL layers will implement a fixed function. The parameters in the CONV/FC layers will be trained with gradient descent so that the class scores that the ConvNet computes are consistent with the labels in the training set for each image.

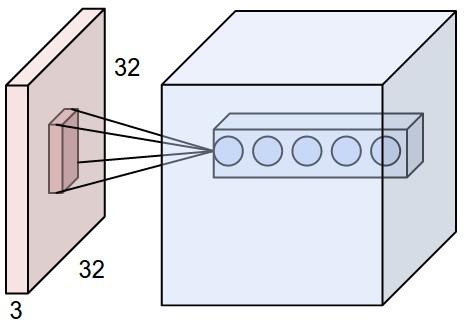
#### Convolutional Layer

The Conv layer is the core building block of a Convolutional Network that does most of the computational heavy lifting.

**Intuition without brain stuff:** The CONV layer’s parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. For example, a typical filter on a first layer of a ConvNet might have size 5x5x3 (i.e. 5 pixels width and height, and 3 because images have depth 3, the color channels). During the forward pass, we slide (more precisely, convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume we will produce a 2-dimensional activation map that gives the responses of that filter at every spatial position. Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network. Now, we will have an entire set of filters in each CONV layer (e.g. 12 filters), and each of them will produce a separate 2-dimensional activation map. We will stack these activation maps along the depth dimension and produce the output volume.

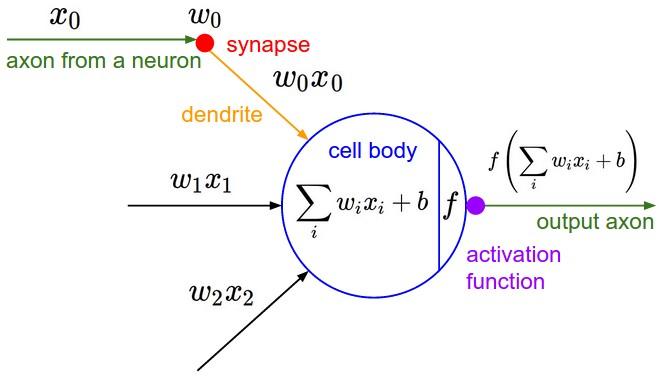
**The brain view:** Every entry in the 3D output volume can also be interpreted as an output of a neuron that looks at only a small region in the input and shares parameters with all neurons to the left and right spatially (since these numbers all result from applying the same filter).

**Local Connectivity:** When dealing with high-dimensional inputs such as images, as we saw above it is impractical to connect neurons to all neurons in the previous volume. Instead, we will connect each neuron to only a local region of the input volume. The spatial extent of this connectivity is a hyper parameter called the **receptive field** of the neuron (equivalently this is the filter size). The extent of the connectivity along the depth axis is always equal to the depth of the input volume. It is important to emphasize again this asymmetry in how we treat the spatial dimensions (width and height) and the depth dimension: The connections are local in space (along width and height), but always full along the entire depth of the input volume.



**Figure 3.3 : Max pooling of spatial dimensions**

An example volume of neurons in the first Convolutional layer. Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all color channels). Note, there are multiple neurons (5 in this example) along the depth, all looking at the same region in the input - see discussion of depth columns in text below.



**Figure 3.4 : Architecture of a neuron in network**

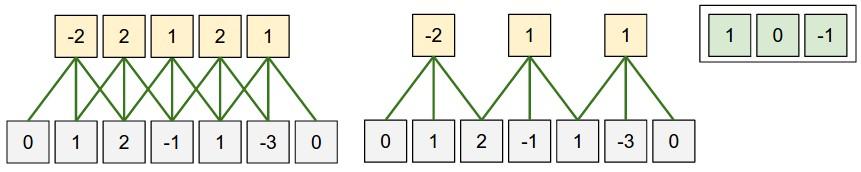
The neurons from the Neural Network chapter remain unchanged: They still compute a dot product of their weights with the input followed by a non-linearity, but their connectivity is now restricted to be local spatially.

**Spatial arrangement**. How many neurons there are in the output volume and how they are arranged are described below:

Three hyper parameters control the size of the output volume: the **depth, stride** and **zero-padding**.

1. First, the **depth** of the output volume is a hyper parameter: it corresponds to the number of filters we would like to use, each learning to look for something different in the input. For example, if the first Convolutional Layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of color. We will refer to a set of neurons that are all looking at the same region of the input as a **depth column** (some people also prefer the term *fibre*).
2. Second, we must specify the **stride** with which we slide the filter. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2 (or uncommonly 3 or more, though this is rare in practice) then the filters jump 2 pixels at a time as we slide them around. This will produce smaller output volumes spatially.
3. As we will soon see, sometimes it will be convenient to pad the input volume with zeros around the border. The size of this **zero-padding** is a hyperparameter. The nice feature of zero padding is that it will allow us to control the spatial size of the output volumes (most commonly as we’ll see soon we will use it to exactly preserve the spatial size of the input volume so the input and output width and height are the same).

We can compute the spatial size of the output volume as a function of the input volume size (WW), the receptive field size of the Conv Layer neurons (FF), the stride with which they are applied (SS), and the amount of zero padding used (PP) on the border. You can convince yourself that the correct formula for calculating how many neurons “fit” is given by (W−F+2P)/S+1(W−F+2P)/S+1. For example for a 7x7 input and a 3x3 filter with stride 1 and pad 0 we would get a 5x5 output. With stride 2 we would get a 3x3 output.



**Figure 3.5 : 1D Convolutions**

In this example there is only one spatial dimension (x-axis), one neuron with a receptive field size of F = 3, the input size is W = 5, and there is zero padding of P = 1. **Left:** The neuron strided across the input in stride of S = 1, giving output of size (5 - 3 + 2)/1+1 = 5. **Right:** The neuron uses stride of S = 2, giving output of size (5 - 3 + 2)/2+1 = 3. Notice that stride S = 3 could not be used since it wouldn't fit neatly across the volume. In terms of the equation, this can be determined since (5-3+2)=4 is not divisible by 3.    
The neuron weights are in this example [1,0,-1] (shown on very right), and its bias is zero. These weights are shared across all yellow neurons.

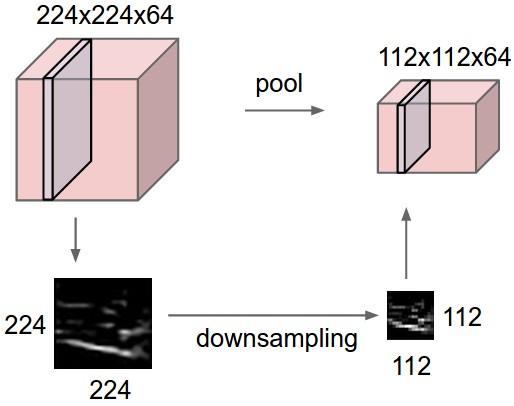
**1x1 convolution**. As an aside, several papers use 1x1 convolutions, as first investigated by [Network in Network](http://arxiv.org/abs/1312.4400). Some people are at first confused to see 1x1 convolutions especially when they come from signal processing background. Normally signals are 2-dimensional so 1x1 convolutions do not make sense (it’s just point wise scaling). However, in ConvNets this is not the case because one must remember that we operate over 3-dimensional volumes, and that the filters always extend through the full depth of the input volume. For example, if the input is [32x32x3] then doing 1x1 convolutions would effectively be doing 3-dimensional dot products (since the input depth is 3 channels).

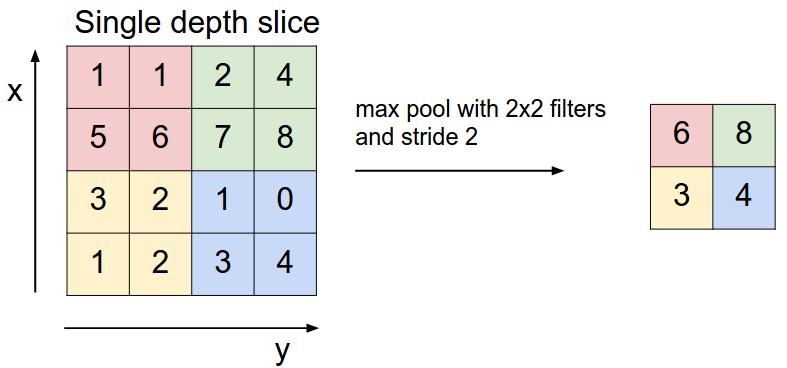
#### Pooling Layer

It is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every MAX operation would in this case be taking a max over 4 numbers (little 2x2 region in some depth slice). The depth dimension remains unchanged. More generally, the pooling layer:

* Accepts a volume of size W1×H1×D1W1×H1×D1
* Requires two hyper parameters:
  + their spatial extent FF,
  + the stride SS,
* Produces a volume of size W2×H2×D2W2×H2×D2 where:
  + W2=(W1−F)/S+1W2=(W1−F)/S+1
  + H2=(H1−F)/S+1H2=(H1−F)/S+1
  + D2=D1D2=D1
* Introduces zero parameters since it computes a fixed function of the input
* Note that it is not common to use zero-padding for Pooling layers.

**General pooling**. In addition to max pooling, the pooling units can also perform other functions, such as *average pooling* or even *L2-norm pooling*. Average pooling was often used historically but has recently fallen out of favor compared to the max pooling operation, which has been shown to work better in practice.





**Figure 3.6 : General Pooling**

In above figures : In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. **Right:** In next figure, The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

**Normalization Layer**

Many types of normalization layers have been proposed for use in ConvNet architectures, sometimes with the intentions of implementing inhibition schemes observed in the biological brain. However, these layers have since fallen out of favor because in practice their contribution has been shown to be minimal, if any.

**Fully-connected layer**

Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset.

**Converting FC layers to CONV layers**

It is worth noting that the only difference between FC and CONV layers is that the neurons in the CONV layer are connected only to a local region in the input, and that many of the neurons in a CONV volume share parameters. However, the neurons in both layers still compute dot products, so their functional form is identical. Therefore, it turns out that it’s possible to convert between FC and CONV layers:

* For any CONV layer there is an FC layer that implements the same forward function. The weight matrix would be a large matrix that is mostly zero except for at certain blocks (due to local connectivity) where the weights in many of the blocks are equal (due to parameter sharing).
* Conversely, any FC layer can be converted to a CONV layer. For example, an FC layer with K=4096K=4096 that is looking at some input volume of size 7×7×5127×7×512 can be equivalently expressed as a CONV layer with F=7,P=0,S=1,K=4096F=7,P=0,S=1,K=4096. In other words, we are setting the filter size to be exactly the size of the input volume, and hence the output will simply be 1×1×40961×1×4096 since only a single depth column “fits” across the input volume, giving identical result as the initial FC layer.

**FC->CONV conversion**. Of these two conversions, the ability to convert an FC layer to a CONV layer is particularly useful in practice. Consider a ConvNet architecture that takes a 224x224x3 image, and then uses a series of CONV layers and POOL layers to reduce the image to an activations volume of size 7x7x512 (in an *AlexNet* architecture that we’ll see later, this is done by use of 5 pooling layers that down sample the input spatially by a factor of two each time, making the final spatial size 224/2/2/2/2/2 = 7). From there, an AlexNet uses two FC layers of size 4096 and finally the last FC layers with 1000 neurons that compute the class scores. We can convert each of these three FC layers to CONV layers as described above:

* Replace the first FC layer that looks at [7x7x512] volume with a CONV layer that uses filter size F=7F=7, giving output volume [1x1x4096].
* Replace the second FC layer with a CONV layer that uses filter size F=1F=1, giving output volume [1x1x4096]
* Replace the last FC layer similarly, with F=1F=1, giving final output [1x1x1000]

Each of these conversions could in practice involve manipulating (e.g. reshaping) the weight matrix WW in each FC layer into CONV layer filters. It turns out that this conversion allows us to “slide” the original ConvNet very efficiently across many spatial positions in a larger image, in a single forward pass.

For example, if 224x224 image gives a volume of size [7x7x512] - i.e. a reduction by 32, then forwarding an image of size 384x384 through the converted architecture would give the equivalent volume in size [12x12x512], since 384/32 = 12. Following through with the next 3 CONV layers that we just converted from FC layers would now give the final volume of size [6x6x1000], since (12 - 7)/1 + 1 = 6. Note that instead of a single vector of class scores of size [1x1x1000], we’re now getting an entire 6x6 array of class scores across the 384x384 image.

Naturally, forwarding the converted ConvNet a single time is much more efficient than iterating the original ConvNet over all those 36 locations, since the 36 evaluations share computation. This trick is often used in practice to get better performance, where for example, it is common to resize an image to make it bigger, use a converted ConvNet to evaluate the class scores at many spatial positions and then average the class scores.

**LSTM Networks**

Long Short Term Memory networks – usually just called “LSTMs” – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by [Hochreiter & Schmidhuber (1997)](http://www.bioinf.jku.at/publications/older/2604.pdf), and were refined and popularized by many people in following work. They work tremendously well on a large variety of problems, and are now widely used.

LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn.

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

Let’s take the example of predicting stock prices for a particular stock. The stock price of today will depend upon:

1. The trend that the stock has been following in the previous days, maybe a downtrend or an uptrend.
2. The price of the stock on the previous day, because many traders compare the stock’s previous day price before buying it.
3. The factors that can affect the price of the stock for today. This can be a new company policy that is being criticized widely, or a drop in the company’s profit, or maybe an unexpected change in the senior leadership of the company.

These dependencies can be generalized to any problem as:

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

**Architecture of LSTMs**

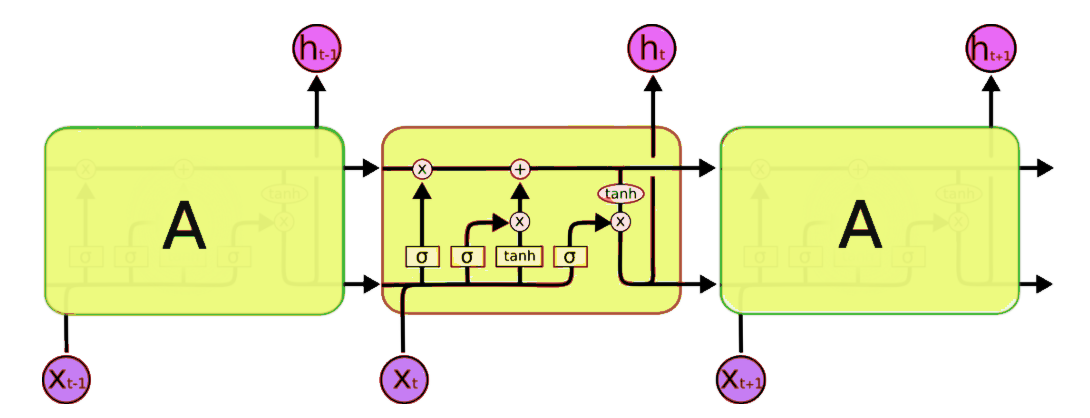
The functioning of LSTM can be visualized by understanding the functioning of a news channel’s team covering a murder story. Now, a news story is built around facts, evidence and statements of many people. Whenever a new event occurs you take either of the three steps.

Let’s say, we were assuming that the murder was done by ‘poisoning’ the victim, but the autopsy report that just came in said that the cause of death was ‘an impact on the head’. Being a part of this news team what do you do? You immediately **forget**the previous cause of death and all stories that were woven around this fact.

What, if an entirely new suspect is introduced into the picture. A person who had grudges with the victim and could be the murderer? You **input**this information into your news feed, right?

Now all these broken pieces of information cannot be served on mainstream media. So, after a certain time interval, you need to summarize this information and **output**the relevant things to your audience. Maybe in the form of “*XYZ turns out to be the prime suspect.*”.

Now the details of the architecture of LSTM network:



**Figure 3.7 : LSTM Network Architecture**

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

**3.2 Extraction of Deep features**

**3.2.1 Inception Algorithm**

Motivation and High Level Considerations : The most straightforward way of improving the performance of deep neural networks is by increasing their size. This includes both increasing the depth – the number of net. This is an easy and safe way of training higher quality models, especially given the availability of a large amount of labeled training data. However, this simple solution comes with two major drawbacks. Bigger size typically means a larger number of parameters, which makes the enlarged network more prone to overfitting, especially if the number of labeled examples in the training set is limited. This is a major bottleneck as strongly labeled datasets are laborious and expensive to obtain, often requiring expert human raters to distinguish between various fine-grained visual categories such as those in ImageNet. The other drawback of uniformly increased network size is the dramatically increased use of computational resources. For example, in a deep vision network, if two convolutional layers are chained, any uniform increase in the number of their filters results in a quadratic increase of computation. If the added capacity is used inefficiently (for example, if most weights end up to be close to zero), then much of the computation is wasted. As the computational budget is always finite, an efficient distribution of computing resources is preferred to an indiscriminate increase of size, even when the main objective is to increase the quality of performance. A fundamental way of solving both of these issues would be to introduce sparsity and replace the fully connected layers by the sparse ones, even inside the convolutions. Main result states that if the probability distribution of the dataset is representable by a large, very sparse deep neural network, then the optimal network topology can be constructed layer after layer by analyzing the correlation statistics of the preceding layer activations and clustering neurons with highly correlated outputs. Although the strict mathematical proof requires very strong conditions, the fact that this statement resonates with the well known Hebbian principle – neurons that fire together, wire together – suggests that the underlying idea is applicable even under less strict conditions, in practice. Unfortunately, today’s computing infrastructures are very inefficient when it comes to numerical calculation on non-uniform sparse data structures. Even if the number of arithmetic operations is reduced by 100×, the overhead of lookups and cache misses would dominate: switching to sparse matrices might not pay off. The gap is widened yet further by the use of steadily improving and highly tuned numerical libraries that allow for extremely fast dense matrix multiplication, exploiting the minute details of the underlying CPU or GPU hardware. Also, non-uniform sparse models require more sophisticated engineering and computing infrastructure. Most current vision oriented machine learning systems utilize sparsity in the spatial domain just by the virtue of employing convolutions. However, convolutions are implemented as collections of dense connections to the patches in the earlier layer. ConvNets have traditionally used random and sparse connection tables in the feature dimensions since in order to break the symmetry and improve learning, yet the trend changed back to full connections with in order to further optimize parallel computation. Current state-of-the-art architectures for computer vision have uniform structure. The large number of filters and greater batch size allows for the efficient use of dense computation. This raises the question of whether there is any hope for a next, intermediate step: an architecture that makes use of filter-level sparsity, as suggested by the theory, but exploits our current hardware by utilizing computations on dense matrices.

The first inception module looks like :

> inception\_3a\_1x1 = Convolution2D(…..)(pool2\_3x3\_s2)

> inception\_3a\_3x3\_reduce = Convolution2D( …..)(pool2\_3x3\_s2)

> inception\_3a\_3x3 = Convolution2D(…..)(inception\_3a\_3x3\_reduce)

> inception\_3a\_5x5\_reduce = Convolution2D(,...)(pool2\_3x3\_s2)

> inception\_3a\_5x5 = Convolution2D(...)(inception\_3a\_5x5\_reduce)

> inception\_3a\_pool = MaxPooling2D(pool\_size=(3,3),....)(pool2\_3x3\_s2)

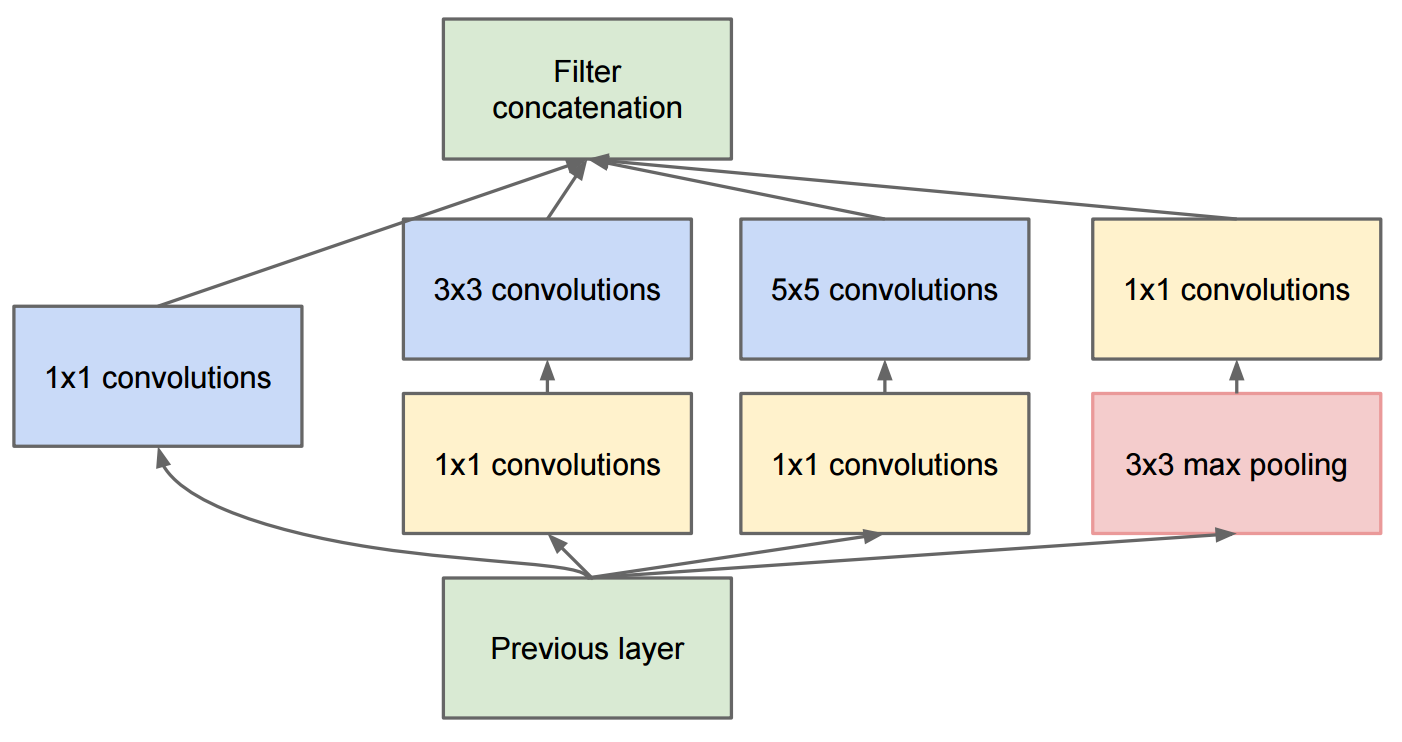
> inception\_3a\_pool\_proj = Convolution2D(,...)(inception\_3a\_pool)

> inception\_3a\_output = merge(**[inception\_3a\_1x1, inception\_3a\_3x3, inception\_3a\_5x5, inception\_3a\_pool\_proj]**, ……..)

The vast literature on sparse matrix computations suggests that clustering sparse matrices into relatively dense submatrices tends to give competitive performance for sparse matrix multiplication. It does not seem far-fetched to think that similar methods would be utilized for the automated construction of non-uniform deep learning architectures in the near future. The Inception architecture started out as a case study for assessing the hypothetical output of a sophisticated network topology construction algorithm that tries to approximate a sparse structure implied by for vision networks and covering the hypothesized outcome by dense, readily available components. With a bit of tuning the gap widened and Inception proved to be especially useful in the context of localization. Interestingly, while most of the original architectural choices have been questioned and tested thoroughly in separation, they turned out to be close to optimal locally. One must be cautious though: although the Inception architecture has become a success for computer vision, it is still questionable whether this can be attributed to the guiding principles that have lead to its construction. Making sure of this would require a much more thorough analysis and verification.

**Architectural Details of Inception**

The main idea of the Inception architecture is to consider how an optimal local sparse structure of a convolutional vision network can be approximated and covered by readily available dense components. Note that assuming translation invariance means that our network will be built from convolutional building blocks. All we need is to find the optimal local construction and to repeat it spatially. Arora et al. suggests a layer-by layer construction where one should analyze the correlation statistics of the last layer and cluster them into groups of units with high correlation. These clusters form the units of the next layer and are connected to the units in the previous layer. We assume that each unit from an earlier layer corresponds to some region of the input image and these units are grouped into filter banks. In the lower layers (the ones close to the input) correlated units would concentrate in local regions. Thus, we would end up with a lot of clusters concentrated in a single region and they can be covered by a layer of 1×1 convolutions in the next layer. However, one can also expect that there will be a smaller number of more spatially spread out clusters that can be covered by convolutions over larger patches, and there will be a decreasing number of patches over larger and larger regions. In order to avoid patch-alignment issues, current incarnations of the Inception architecture are restricted to filter sizes 1×1, 3×3 and 5×5; this decision was based more on convenience rather than necessity.



**Figure 3.8 : Inception model Architecture**

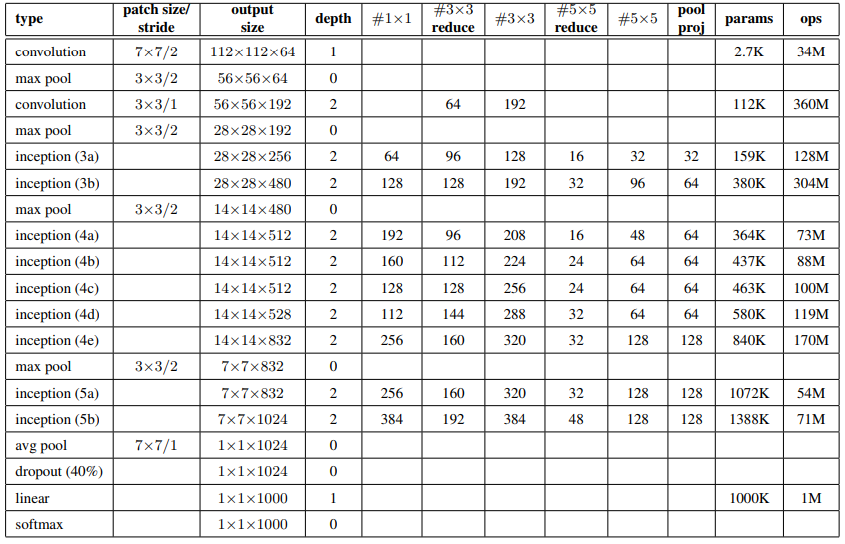
It also means that the suggested architecture is a combination of all those layers with their output filter banks concatenated into a single output vector forming the input of the next stage. Additionally, since pooling operations have been essential for the success of current convolutional networks, it suggests that adding an alternative parallel pooling path in each such stage should have additional beneficial effect, too. As these “Inception modules” are stacked on top of each other, their output correlation statistics are bound to vary: as features of higher abstraction are captured by higher layers, their spatial concentration is expected to decrease. This suggests that the ratio of 3×3 and 5×5 convolutions should increase as we move to higher layers. One big problem with the above modules, at least in this na¨ıve form, is that even a modest number of 5×5 convolutions can be prohibitively expensive on top of a convolutional layer with a large number of filters. This problem becomes even more pronounced once pooling units are added to the mix: the number of output filters equals to the number of filters in the previous stage. The merging of output of the pooling layer with outputs of the convolutional layers would lead to an inevitable increase in the number of outputs from stage to stage. While this architecture might cover the optimal sparse structure, it would do it very inefficiently, leading to a computational blow up within a few stages. This leads to the second idea of the Inception architecture: judiciously reducing dimension wherever the computational requirements would increase too much otherwise. This is based on the success of embeddings: even low dimensional embeddings might contain a lot of information about a relatively large image patch. However, embeddings represent information in a dense, compressed form and compressed information is harder to process. The representation should be kept sparse at most places and compress the signals only whenever they have to be aggregated en masse. That is, 1×1 convolutions are used to compute reductions before the expensive 3×3 and 5×5 convolutions. Besides being used as reductions, they also include the use of rectified linear activation making them dual-purpose. In general, an Inception network is a network consisting of modules of the above type stacked upon each other, with occasional max-pooling layers with stride 2 to halve the resolution of the grid. For technical reasons (memory efficiency during training), it seemed beneficial to start using Inception modules only at higher layers while keeping the lower layers in traditional convolutional fashion. This is not strictly necessary, simply reflecting some infrastructural inefficiencies in our current implementation. A useful aspect of this architecture is that it allows for increasing the number of units at each stage significantly without an uncontrolled blow-up in computational complexity at later stages. This is achieved by the ubiquitous use of dimensionality reduction prior to expensive convolutions with larger patch sizes. Furthermore, the design follows the practical intuition that visual information should be processed at various scales and then aggregated so that the next stage can abstract features from the different scales simultaneously. The improved use of computational resources allows for increasing both the width of each stage as well as the number of stages without getting into computational difficulties. One can utilize the Inception architecture to create slightly inferior, but computationally cheaper versions of it. We have found that all the available knobs and levers allow for a controlled balancing of computational resources resulting in networks that are 3 − 10× faster than similarly performing networks with non-Inception architecture.

**3.2.2 GoogLeNet**

All the convolutions, including those inside the Inception modules, use rectified linear activation. The size of the receptive field in our network is 224×224 in the RGB color space with zero mean. “3×3 reduce” and “5×5 reduce” stands for the number of 1×1 filters in the reduction layer used before the 3×3 and 5×5 convolutions. One can see the number of 1×1 filters in the projection layer after the built-in max-pooling in the pool proj column. All these reduction/projection layers use rectified linear activation as well.

The network is 22 layers deep when counting only layers with parameters (or 27 layers if we also count pooling). The overall number of layers (independent building blocks) used for the construction of the network is about 100. The exact number depends on how layers are counted by the machine learning infrastructure. The use of average pooling before the classifier is based on, although our implementation has an additional linear layer. The linear layer enables us to easily adapt our networks to other label sets, however it is used mostly for convenience and we do not expect it to have a major effect. We found that a move from fully connected layers to average pooling improved the top-1 accuracy by about 0.6%, however the use of dropout remained essential even after removing the fully connected layers. Given relatively large depth of the network, the ability to propagate gradients back through all the layers in an effective manner was a concern. The strong performance of shallower networks

**Table 3 : GoogLeNet Layers**



on this task suggests that the features produced by the layers in the middle of the network should be very discriminative. By adding auxiliary classifiers connected to these intermediate layers, discrimination in the lower stages in the classifier was expected. This was thought to combat the vanishing gradient problem while providing regularization. These classifiers take the form of smaller convolutional networks put on top of the output of the Inception (4a) and (4d) modules. During training, their loss gets added to the total loss of the network with a discount weight (the losses of the auxiliary classifiers were weighted by 0.3).

At inference time, these auxiliary networks are discarded. Later control experiments have shown that the effect of the auxiliary networks is relatively minor (around 0.5%) and that it required only one of them to achieve the same effect.

The exact structure of the extra network on the side, including the auxiliary classifier, is as follows:

* An average pooling layer with 5×5 filter size and stride 3, resulting in an 4×4×512 output for the (4a), and 4×4×528 for the (4d) stage.
* A 1×1 convolution with 128 filters for dimension reduction and rectified linear activation.
* A fully connected layer with 1024 units and rectified linear activation.
* A dropout layer with 70% ratio of dropped outputs.
* A linear layer with softmax loss as the classifier (predicting the same 1000 classes as the main classifier, but removed at inference time).

**4. Implementation**

**4.1 Code**

The project is a simulation of existing research work on “Going Deeper with Convolutions”. The research is about implementation of deeper and efficient neural network called ‘Inception’. Inception module is network inside a network. GoogLeNet architecture is implemented using Inception network. That’s why GoogLeNet has won Image Net Challenge and attained less error rate of about 6% only. Now, all that matters is the accuracy. Hence, to simulate the proposed results lively, we implemented the following code with implementation of GoogLeNet architecture in Keras model to predict the outputs of a given image.

The implemented code for observing the deep features of input image is :

#googlenet.py

from scipy.misc import imread, imresize

from keras.layers import Input, Dense, Conv2D, MaxPooling2D, AveragePooling2D, ZeroPadding2D, Dropout, Flatten, merge, Reshape, Activation

from keras.models import Model

from keras.regularizers import l2

from keras.optimizers import SGD

from keras.utils import np\_utils

from keras.callbacks import ReduceLROnPlateau, CSVLogger, EarlyStopping, ModelCheckpoint

from googlenet\_custom\_layers import PoolHelper,LRN

import numpy as np

import os

import json

import re

def create\_googlenet(weights\_path=None):

input = Input(shape=(3, 224, 224))

conv1\_7x7\_s2 = Conv2D(64,7,7,subsample=(2,2),border\_mode='same',activation='relu',name='conv1/7x7\_s2',W\_regularizer=l2(0.0002))(input)

conv1\_zero\_pad = ZeroPadding2D(padding=(1, 1))(conv1\_7x7\_s2)

pool1\_helper = PoolHelper()(conv1\_zero\_pad)

pool1\_3x3\_s2 = MaxPooling2D(pool\_size=(3,3),strides=(2,2),border\_mode='valid',name='pool1/3x3\_s2')(pool1\_helper)

pool1\_norm1 = LRN(name='pool1/norm1')(pool1\_3x3\_s2)

conv2\_3x3\_reduce = Conv2D(64,1,1,border\_mode='same',activation='relu',name='conv2/3x3\_reduce',W\_regularizer=l2(0.0002))(pool1\_norm1)

conv2\_3x3 = Conv2D(192,3,3,border\_mode='same',activation='relu',name='conv2/3x3',W\_regularizer=l2(0.0002))(conv2\_3x3\_reduce)

conv2\_norm2 = LRN(name='conv2/norm2')(conv2\_3x3)

conv2\_zero\_pad = ZeroPadding2D(padding=(1, 1))(conv2\_norm2)

pool2\_helper = PoolHelper()(conv2\_zero\_pad)

pool2\_3x3\_s2 = MaxPooling2D(pool\_size=(3,3),strides=(2,2),border\_mode='valid',name='pool2/3x3\_s2')(pool2\_helper)

inception\_3a\_1x1 = Conv2D(64,1,1,border\_mode='same',activation='relu',name='inception\_3a/1x1',W\_regularizer=l2(0.0002))(pool2\_3x3\_s2)

inception\_3a\_3x3\_reduce = Conv2D(96,1,1,border\_mode='same',activation='relu',name='inception\_3a/3x3\_reduce',W\_regularizer=l2(0.0002))(pool2\_3x3\_s2)

inception\_3a\_3x3 = Conv2D(128,3,3,border\_mode='same',activation='relu',name='inception\_3a/3x3',W\_regularizer=l2(0.0002))(inception\_3a\_3x3\_reduce)

inception\_3a\_5x5\_reduce = Conv2D(16,1,1,border\_mode='same',activation='relu',name='inception\_3a/5x5\_reduce',W\_regularizer=l2(0.0002))(pool2\_3x3\_s2)

inception\_3a\_5x5 = Conv2D(32,5,5,border\_mode='same',activation='relu',name='inception\_3a/5x5',W\_regularizer=l2(0.0002))(inception\_3a\_5x5\_reduce)

inception\_3a\_pool = MaxPooling2D(pool\_size=(3,3),strides=(1,1),border\_mode='same',name='inception\_3a/pool')(pool2\_3x3\_s2)

inception\_3a\_pool\_proj = Conv2D(32,1,1,border\_mode='same',activation='relu',name='inception\_3a/pool\_proj',W\_regularizer=l2(0.0002))(inception\_3a\_pool)

inception\_3a\_output = merge([inception\_3a\_1x1,inception\_3a\_3x3,inception\_3a\_5x5,inception\_3a\_pool\_proj],mode='concat',concat\_axis=1,name='inception\_3a/output')

inception\_3b\_1x1 = Conv2D(128,1,1,border\_mode='same',activation='relu',name='inception\_3b/1x1',W\_regularizer=l2(0.0002))(inception\_3a\_output)

inception\_3b\_3x3\_reduce = Conv2D(128,1,1,border\_mode='same',activation='relu',name='inception\_3b/3x3\_reduce',W\_regularizer=l2(0.0002))(inception\_3a\_output)

inception\_3b\_3x3 = Conv2D(192,3,3,border\_mode='same',activation='relu',name='inception\_3b/3x3',W\_regularizer=l2(0.0002))(inception\_3b\_3x3\_reduce)

inception\_3b\_5x5\_reduce = Conv2D(32,1,1,border\_mode='same',activation='relu',name='inception\_3b/5x5\_reduce',W\_regularizer=l2(0.0002))(inception\_3a\_output)

inception\_3b\_5x5 = Conv2D(96,5,5,border\_mode='same',activation='relu',name='inception\_3b/5x5',W\_regularizer=l2(0.0002))(inception\_3b\_5x5\_reduce)

inception\_3b\_pool = MaxPooling2D(pool\_size=(3,3),strides=(1,1),border\_mode='same',name='inception\_3b/pool')(inception\_3a\_output)

inception\_3b\_pool\_proj = Conv2D(64,1,1,border\_mode='same',activation='relu',name='inception\_3b/pool\_proj',W\_regularizer=l2(0.0002))(inception\_3b\_pool)

inception\_3b\_output = merge([inception\_3b\_1x1,inception\_3b\_3x3,inception\_3b\_5x5,inception\_3b\_pool\_proj],mode='concat',concat\_axis=1,name='inception\_3b/output')

inception\_3b\_output\_zero\_pad = ZeroPadding2D(padding=(1, 1))(inception\_3b\_output)

pool3\_helper = PoolHelper()(inception\_3b\_output\_zero\_pad)

pool3\_3x3\_s2 = MaxPooling2D(pool\_size=(3,3),strides=(2,2),border\_mode='valid',name='pool3/3x3\_s2')(pool3\_helper)

inception\_4a\_1x1 = Conv2D(192,1,1,border\_mode='same',activation='relu',name='inception\_4a/1x1',W\_regularizer=l2(0.0002))(pool3\_3x3\_s2)

inception\_4a\_3x3\_reduce = Conv2D(96,1,1,border\_mode='same',activation='relu',name='inception\_4a/3x3\_reduce',W\_regularizer=l2(0.0002))(pool3\_3x3\_s2)

inception\_4a\_3x3 = Conv2D(208,3,3,border\_mode='same',activation='relu',name='inception\_4a/3x3',W\_regularizer=l2(0.0002))(inception\_4a\_3x3\_reduce)

inception\_4a\_5x5\_reduce = Conv2D(16,1,1,border\_mode='same',activation='relu',name='inception\_4a/5x5\_reduce',W\_regularizer=l2(0.0002))(pool3\_3x3\_s2)

inception\_4a\_5x5 = Conv2D(48,5,5,border\_mode='same',activation='relu',name='inception\_4a/5x5',W\_regularizer=l2(0.0002))(inception\_4a\_5x5\_reduce)

inception\_4a\_pool = MaxPooling2D(pool\_size=(3,3),strides=(1,1),border\_mode='same',name='inception\_4a/pool')(pool3\_3x3\_s2)

inception\_4a\_pool\_proj = Conv2D(64,1,1,border\_mode='same',activation='relu',name='inception\_4a/pool\_proj',W\_regularizer=l2(0.0002))(inception\_4a\_pool)

inception\_4a\_output = merge([inception\_4a\_1x1,inception\_4a\_3x3,inception\_4a\_5x5,inception\_4a\_pool\_proj],mode='concat',concat\_axis=1,name='inception\_4a/output')

loss1\_ave\_pool = AveragePooling2D(pool\_size=(5,5),strides=(3,3),name='loss1/ave\_pool')(inception\_4a\_output)

loss1\_conv = Conv2D(128,1,1,border\_mode='same',activation='relu',name='loss1/conv',W\_regularizer=l2(0.0002))(loss1\_ave\_pool)

loss1\_flat = Flatten()(loss1\_conv)

loss1\_fc = Dense(1024,activation='relu',name='loss1/fc',W\_regularizer=l2(0.0002))(loss1\_flat)

loss1\_drop\_fc = Dropout(0.7)(loss1\_fc)

loss1\_classifier = Dense(1000,name='loss1/classifier',W\_regularizer=l2(0.0002))(loss1\_drop\_fc)

loss1\_classifier\_act = Activation('softmax')(loss1\_classifier)

inception\_4b\_1x1 = Conv2D(160,1,1,border\_mode='same',activation='relu',name='inception\_4b/1x1',W\_regularizer=l2(0.0002))(inception\_4a\_output)

inception\_4b\_3x3\_reduce = Conv2D(112,1,1,border\_mode='same',activation='relu',name='inception\_4b/3x3\_reduce',W\_regularizer=l2(0.0002))(inception\_4a\_output)

inception\_4b\_3x3 = Conv2D(224,3,3,border\_mode='same',activation='relu',name='inception\_4b/3x3',W\_regularizer=l2(0.0002))(inception\_4b\_3x3\_reduce)

inception\_4b\_5x5\_reduce = Conv2D(24,1,1,border\_mode='same',activation='relu',name='inception\_4b/5x5\_reduce',W\_regularizer=l2(0.0002))(inception\_4a\_output)

inception\_4b\_5x5 = Conv2D(64,5,5,border\_mode='same',activation='relu',name='inception\_4b/5x5',W\_regularizer=l2(0.0002))(inception\_4b\_5x5\_reduce)

inception\_4b\_pool = MaxPooling2D(pool\_size=(3,3),strides=(1,1),border\_mode='same',name='inception\_4b/pool')(inception\_4a\_output)

inception\_4b\_pool\_proj = Conv2D(64,1,1,border\_mode='same',activation='relu',name='inception\_4b/pool\_proj',W\_regularizer=l2(0.0002))(inception\_4b\_pool)

inception\_4b\_output = merge([inception\_4b\_1x1,inception\_4b\_3x3,inception\_4b\_5x5,inception\_4b\_pool\_proj],mode='concat',concat\_axis=1,name='inception\_4b\_output')

inception\_4c\_1x1 = Conv2D(128,1,1,border\_mode='same',activation='relu',name='inception\_4c/1x1',W\_regularizer=l2(0.0002))(inception\_4b\_output)

inception\_4c\_3x3\_reduce = Conv2D(128,1,1,border\_mode='same',activation='relu',name='inception\_4c/3x3\_reduce',W\_regularizer=l2(0.0002))(inception\_4b\_output)

inception\_4c\_3x3 = Conv2D(256,3,3,border\_mode='same',activation='relu',name='inception\_4c/3x3',W\_regularizer=l2(0.0002))(inception\_4c\_3x3\_reduce)

inception\_4c\_5x5\_reduce = Conv2D(24,1,1,border\_mode='same',activation='relu',name='inception\_4c/5x5\_reduce',W\_regularizer=l2(0.0002))(inception\_4b\_output)

inception\_4c\_5x5 = Conv2D(64,5,5,border\_mode='same',activation='relu',name='inception\_4c/5x5',W\_regularizer=l2(0.0002))(inception\_4c\_5x5\_reduce)

inception\_4c\_pool = MaxPooling2D(pool\_size=(3,3),strides=(1,1),border\_mode='same',name='inception\_4c/pool')(inception\_4b\_output)

inception\_4c\_pool\_proj = Conv2D(64,1,1,border\_mode='same',activation='relu',name='inception\_4c/pool\_proj',W\_regularizer=l2(0.0002))(inception\_4c\_pool)

inception\_4c\_output = merge([inception\_4c\_1x1,inception\_4c\_3x3,inception\_4c\_5x5,inception\_4c\_pool\_proj],mode='concat',concat\_axis=1,name='inception\_4c/output')

inception\_4d\_1x1 = Conv2D(112,1,1,border\_mode='same',activation='relu',name='inception\_4d/1x1',W\_regularizer=l2(0.0002))(inception\_4c\_output)

inception\_4d\_3x3\_reduce = Conv2D(144,1,1,border\_mode='same',activation='relu',name='inception\_4d/3x3\_reduce',W\_regularizer=l2(0.0002))(inception\_4c\_output)

inception\_4d\_3x3 = Conv2D(288,3,3,border\_mode='same',activation='relu',name='inception\_4d/3x3',W\_regularizer=l2(0.0002))(inception\_4d\_3x3\_reduce)

inception\_4d\_5x5\_reduce = Conv2D(32,1,1,border\_mode='same',activation='relu',name='inception\_4d/5x5\_reduce',W\_regularizer=l2(0.0002))(inception\_4c\_output)

inception\_4d\_5x5 = Conv2D(64,5,5,border\_mode='same',activation='relu',name='inception\_4d/5x5',W\_regularizer=l2(0.0002))(inception\_4d\_5x5\_reduce)

inception\_4d\_pool = MaxPooling2D(pool\_size=(3,3),strides=(1,1),border\_mode='same',name='inception\_4d/pool')(inception\_4c\_output)

inception\_4d\_pool\_proj = Conv2D(64,1,1,border\_mode='same',activation='relu',name='inception\_4d/pool\_proj',W\_regularizer=l2(0.0002))(inception\_4d\_pool)

inception\_4d\_output = merge([inception\_4d\_1x1,inception\_4d\_3x3,inception\_4d\_5x5,inception\_4d\_pool\_proj],mode='concat',concat\_axis=1,name='inception\_4d/output')

loss2\_ave\_pool = AveragePooling2D(pool\_size=(5,5),strides=(3,3),name='loss2/ave\_pool')(inception\_4d\_output)

loss2\_conv = Conv2D(128,1,1,border\_mode='same',activation='relu',name='loss2/conv',W\_regularizer=l2(0.0002))(loss2\_ave\_pool)

loss2\_flat = Flatten()(loss2\_conv)

loss2\_fc = Dense(1024,activation='relu',name='loss2/fc',W\_regularizer=l2(0.0002))(loss2\_flat)

loss2\_drop\_fc = Dropout(0.7)(loss2\_fc)

loss2\_classifier = Dense(1000,name='loss2/classifier',W\_regularizer=l2(0.0002))(loss2\_drop\_fc)

loss2\_classifier\_act = Activation('softmax')(loss2\_classifier)

inception\_4e\_1x1 = Conv2D(256,1,1,border\_mode='same',activation='relu',name='inception\_4e/1x1',W\_regularizer=l2(0.0002))(inception\_4d\_output)

inception\_4e\_3x3\_reduce = Conv2D(160,1,1,border\_mode='same',activation='relu',name='inception\_4e/3x3\_reduce',W\_regularizer=l2(0.0002))(inception\_4d\_output)

inception\_4e\_3x3 = Conv2D(320,3,3,border\_mode='same',activation='relu',name='inception\_4e/3x3',W\_regularizer=l2(0.0002))(inception\_4e\_3x3\_reduce)

inception\_4e\_5x5\_reduce = Conv2D(32,1,1,border\_mode='same',activation='relu',name='inception\_4e/5x5\_reduce',W\_regularizer=l2(0.0002))(inception\_4d\_output)

inception\_4e\_5x5 = Conv2D(128,5,5,border\_mode='same',activation='relu',name='inception\_4e/5x5',W\_regularizer=l2(0.0002))(inception\_4e\_5x5\_reduce)

inception\_4e\_pool = MaxPooling2D(pool\_size=(3,3),strides=(1,1),border\_mode='same',name='inception\_4e/pool')(inception\_4d\_output)

inception\_4e\_pool\_proj = Conv2D(128,1,1,border\_mode='same',activation='relu',name='inception\_4e/pool\_proj',W\_regularizer=l2(0.0002))(inception\_4e\_pool)

inception\_4e\_output = merge([inception\_4e\_1x1,inception\_4e\_3x3,inception\_4e\_5x5,inception\_4e\_pool\_proj],mode='concat',concat\_axis=1,name='inception\_4e/output')

inception\_4e\_output\_zero\_pad = ZeroPadding2D(padding=(1, 1))(inception\_4e\_output)

pool4\_helper = PoolHelper()(inception\_4e\_output\_zero\_pad)

pool4\_3x3\_s2 = MaxPooling2D(pool\_size=(3,3),strides=(2,2),border\_mode='valid',name='pool4/3x3\_s2')(pool4\_helper)

inception\_5a\_1x1 = Conv2D(256,1,1,border\_mode='same',activation='relu',name='inception\_5a/1x1',W\_regularizer=l2(0.0002))(pool4\_3x3\_s2)

inception\_5a\_3x3\_reduce = Conv2D(160,1,1,border\_mode='same',activation='relu',name='inception\_5a/3x3\_reduce',W\_regularizer=l2(0.0002))(pool4\_3x3\_s2)

inception\_5a\_3x3 = Conv2D(320,3,3,border\_mode='same',activation='relu',name='inception\_5a/3x3',W\_regularizer=l2(0.0002))(inception\_5a\_3x3\_reduce)

inception\_5a\_5x5\_reduce = Conv2D(32,1,1,border\_mode='same',activation='relu',name='inception\_5a/5x5\_reduce',W\_regularizer=l2(0.0002))(pool4\_3x3\_s2)

inception\_5a\_5x5 = Conv2D(128,5,5,border\_mode='same',activation='relu',name='inception\_5a/5x5',W\_regularizer=l2(0.0002))(inception\_5a\_5x5\_reduce)

inception\_5a\_pool = MaxPooling2D(pool\_size=(3,3),strides=(1,1),border\_mode='same',name='inception\_5a/pool')(pool4\_3x3\_s2)

inception\_5a\_pool\_proj = Conv2D(128,1,1,border\_mode='same',activation='relu',name='inception\_5a/pool\_proj',W\_regularizer=l2(0.0002))(inception\_5a\_pool)

inception\_5a\_output = merge([inception\_5a\_1x1,inception\_5a\_3x3,inception\_5a\_5x5,inception\_5a\_pool\_proj],mode='concat',concat\_axis=1,name='inception\_5a/output')

inception\_5b\_1x1 = Conv2D(384,1,1,border\_mode='same',activation='relu',name='inception\_5b/1x1',W\_regularizer=l2(0.0002))(inception\_5a\_output)

inception\_5b\_3x3\_reduce = Conv2D(192,1,1,border\_mode='same',activation='relu',name='inception\_5b/3x3\_reduce',W\_regularizer=l2(0.0002))(inception\_5a\_output)

inception\_5b\_3x3 = Conv2D(384,3,3,border\_mode='same',activation='relu',name='inception\_5b/3x3',W\_regularizer=l2(0.0002))(inception\_5b\_3x3\_reduce)

inception\_5b\_5x5\_reduce = Conv2D(48,1,1,border\_mode='same',activation='relu',name='inception\_5b/5x5\_reduce',W\_regularizer=l2(0.0002))(inception\_5a\_output)

inception\_5b\_5x5 = Conv2D(128,5,5,border\_mode='same',activation='relu',name='inception\_5b/5x5',W\_regularizer=l2(0.0002))(inception\_5b\_5x5\_reduce)

inception\_5b\_pool = MaxPooling2D(pool\_size=(3,3),strides=(1,1),border\_mode='same',name='inception\_5b/pool')(inception\_5a\_output)

inception\_5b\_pool\_proj = Conv2D(128,1,1,border\_mode='same',activation='relu',name='inception\_5b/pool\_proj',W\_regularizer=l2(0.0002))(inception\_5b\_pool)

inception\_5b\_output = merge([inception\_5b\_1x1,inception\_5b\_3x3,inception\_5b\_5x5,inception\_5b\_pool\_proj],mode='concat',concat\_axis=1,name='inception\_5b/output')

pool5\_7x7\_s1 = AveragePooling2D(pool\_size=(7,7),strides=(1,1),name='pool5/7x7\_s2')(inception\_5b\_output)

loss3\_flat = Flatten()(pool5\_7x7\_s1)

pool5\_drop\_7x7\_s1 = Dropout(0.4)(loss3\_flat)

loss3\_classifier = Dense(1000,name='loss3/classifier',W\_regularizer=l2(0.0002))(pool5\_drop\_7x7\_s1)

loss3\_classifier\_act = Activation('softmax',name='prob')(loss3\_classifier)

googlenet = Model(input=input, output=[loss1\_classifier\_act,loss2\_classifier\_act,loss3\_classifier\_act])

if weights\_path:

googlenet.load\_weights(weights\_path)

return googlenet

if \_\_name\_\_ == "\_\_main\_\_":

#train\_images = np.load('train\_images.npy')

#train\_labels = np.load('train\_labels.npy')

lr\_reducer = ReduceLROnPlateau(factor = np.sqrt(0.1), cooldown=0, patience=2, min\_lr=0.5e-6)

csv\_logger = CSVLogger('Googlenet.csv')

early\_stopper = EarlyStopping(min\_delta=0.001,patience=30)

model\_checkpoint = ModelCheckpoint('Googlenet.hdf5',monitor = 'val\_loss', verbose = 1,save\_best\_only=True)

num\_classes = 10

#train\_labels = np\_utils.to\_categorical(train\_labels,num\_classes)

model = create\_googlenet('googlenet\_weights.h5')

print(len(model.layers))

sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)

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

img = imresize(imread('rose.jpg', mode='RGB'), (224, 224)).astype(np.float32)

img[:, :, 0] -= 123.68

img[:, :, 1] -= 116.779

img[:, :, 2] -= 103.939

img[:,:,[0,1,2]] = img[:,:,[2,1,0]]

img = img.transpose((2, 0, 1))

img = np.expand\_dims(img, axis=0)

out = model.predict(img) # note: the model has three outputs

label = np.argmax(out[2])

print label

text\_file = open("labels.txt", "r")

lines = text\_file.read().split('\n')

lines = lines[0:len(lines)-1]

labels\_list = {}

for line in lines:

string = re.split(r'\t+',line)

imagename = string[0]

label1 = string[1]

#print(type(imagename))

#print label

labels\_list[imagename]=label1

wnet\_text = open("words.txt","r")

lines = wnet\_text.read().split('\n')

wnet\_text\_dict = {}

for line in lines:

string = re.split(r'\t+',line)

wnet = string[0]

text = string[1]

wnet\_text\_dict[wnet]=text

image\_text = {}

for key,value in labels\_list.iteritems():

image\_text[key] = wnet\_text\_dict[value]

#print (str(key)+'\t'+str(image\_text[key]))

final\_labels = {}

final\_text = open("finallabels.txt","r")

lines = final\_text.read().split('\n')

lines = lines[0:len(lines)-1]

for line in lines:

string = re.split(r'\s+',line)

#print(string)

final\_labels[string[0]]=string[1]

imagenet\_labels = {0: 'tench, Tinca tinca',

1: 'goldfish, Carassius auratus',

2: 'great white shark, white shark, man-eater, man-eating shark, Carcharodon carcharias',

3: 'tiger shark, Galeocerdo cuvieri',

4: 'hammerhead, hammerhead shark',

5: 'electric ray, crampfish, numbfish, torpedo',

6: 'stingray',

7: 'cock',

8: 'hen',

9: 'ostrich, Struthio camelus',

10: 'brambling, Fringilla montifringilla',

11: 'goldfinch, Carduelis carduelis',

12: 'house finch, linnet, Carpodacus mexicanus',

13: 'junco, snowbird',

14: 'indigo bunting, indigo finch, indigo bird, Passerina cyanea',

15: 'robin, American robin, Turdus migratorius',

16: 'bulbul',

17: 'jay',

18: 'magpie',

19: 'chickadee',

20: 'water ouzel, dipper',

21: 'kite',

22: 'bald eagle, American eagle, Haliaeetus leucocephalus',

23: 'vulture',

24: 'great grey owl, great gray owl, Strix nebulosa',

25: 'European fire salamander, Salamandra salamandra',

26: 'common newt, Triturus vulgaris',

27: 'eft',

28: 'spotted salamander, Ambystoma maculatum',

29: 'axolotl, mud puppy, Ambystoma mexicanum',

30: 'bullfrog, Rana catesbeiana',

31: 'tree frog, tree-frog',

32: 'tailed frog, bell toad, ribbed toad, tailed toad, Ascaphus trui',

33: 'loggerhead, loggerhead turtle, Caretta caretta',

34: 'leatherback turtle, leatherback, leathery turtle, Dermochelys coriacea',

35: 'mud turtle',

36: 'terrapin',

37: 'box turtle, box tortoise',

38: 'banded gecko',

39: 'common iguana, iguana, Iguana iguana',

40: 'American chameleon, anole, Anolis carolinensis',

41: 'whiptail, whiptail lizard',

42: 'agama',

43: 'frilled lizard, Chlamydosaurus kingi',

44: 'alligator lizard',

45: 'Gila monster, Heloderma suspectum',

46: 'green lizard, Lacerta viridis',

47: 'African chameleon, Chamaeleo chamaeleon',

48: 'Komodo dragon, Komodo lizard, dragon lizard, giant lizard, Varanus komodoensis',

49: 'African crocodile, Nile crocodile, Crocodylus niloticus',

50: 'American alligator, Alligator mississipiensis',

51: 'triceratops',

52: 'thunder snake, worm snake, Carphophis amoenus',

53: 'ringneck snake, ring-necked snake, ring snake',

54: 'hognose snake, puff adder, sand viper',

55: 'green snake, grass snake',

56: 'king snake, kingsnake',

57: 'garter snake, grass snake',

58: 'water snake',

59: 'vine snake',

60: 'night snake, Hypsiglena torquata',

61: 'boa constrictor, Constrictor constrictor',

62: 'rock python, rock snake, Python sebae',

63: 'Indian cobra, Naja naja',

64: 'green mamba',

65: 'sea snake',

66: 'horned viper, cerastes, sand viper, horned asp, Cerastes cornutus',

67: 'diamondback, diamondback rattlesnake, Crotalus adamanteus',

68: 'sidewinder, horned rattlesnake, Crotalus cerastes',

69: 'trilobite',

70: 'harvestman, daddy longlegs, Phalangium opilio',

71: 'scorpion',

72: 'black and gold garden spider, Argiope aurantia',

73: 'barn spider, Araneus cavaticus',

74: 'garden spider, Aranea diademata',

75: 'black widow, Latrodectus mactans',

76: 'tarantula',

77: 'wolf spider, hunting spider',

78: 'tick',

79: 'centipede',

80: 'black grouse',

81: 'ptarmigan',

82: 'ruffed grouse, partridge, Bonasa umbellus',

83: 'prairie chicken, prairie grouse, prairie fowl',

84: 'peacock',

85: 'quail',

86: 'partridge',

87: 'African grey, African gray, Psittacus erithacus',

88: 'macaw',

89: 'sulphur-crested cockatoo, Kakatoe galerita, Cacatua galerita',

90: 'lorikeet',

91: 'coucal',

92: 'bee eater',

93: 'hornbill',

94: 'hummingbird',

95: 'jacamar',

96: 'toucan',

97: 'drake',

98: 'red-breasted merganser, Mergus serrator',

99: 'goose',

100: 'black swan, Cygnus atratus',

101: 'tusker',

102: 'echidna, spiny anteater, anteater',

103: 'platypus, duckbill, duckbilled platypus, duck-billed platypus, Ornithorhynchus anatinus',

104: 'wallaby, brush kangaroo',

105: 'koala, koala bear, kangaroo bear, native bear, Phascolarctos cinereus',

106: 'wombat',

107: 'jellyfish',

108: 'sea anemone, anemone',

109: 'brain coral',

110: 'flatworm, platyhelminth',

111: 'nematode, nematode worm, roundworm',

112: 'conch',

113: 'snail',

114: 'slug',

115: 'sea slug, nudibranch',

116: 'chiton, coat-of-mail shell, sea cradle, polyplacophore',

117: 'chambered nautilus, pearly nautilus, nautilus',

118: 'Dungeness crab, Cancer magister',

119: 'rock crab, Cancer irroratus',

120: 'fiddler crab',

121: 'king crab, Alaska crab, Alaskan king crab, Alaska king crab, Paralithodes camtschatica',

122: 'American lobster, Northern lobster, Maine lobster, Homarus americanus',

123: 'spiny lobster, langouste, rock lobster, crawfish, crayfish, sea crawfish',

124: 'crayfish, crawfish, crawdad, crawdaddy',

125: 'hermit crab',

126: 'isopod',

127: 'white stork, Ciconia ciconia',

128: 'black stork, Ciconia nigra',

129: 'spoonbill',

130: 'flamingo',

131: 'little blue heron, Egretta caerulea',

132: 'American egret, great white heron, Egretta albus',

133: 'bittern',

134: 'crane',

135: 'limpkin, Aramus pictus',

136: 'European gallinule, Porphyrio porphyrio',

137: 'American coot, marsh hen, mud hen, water hen, Fulica americana',

138: 'bustard',

139: 'ruddy turnstone, Arenaria interpres',

140: 'red-backed sandpiper, dunlin, Erolia alpina',

141: 'redshank, Tringa totanus',

142: 'dowitcher',

143: 'oystercatcher, oyster catcher',

144: 'pelican',

145: 'king penguin, Aptenodytes patagonica',

146: 'albatross, mollymawk',

147: 'grey whale, gray whale, devilfish, Eschrichtius gibbosus, Eschrichtius robustus',

148: 'killer whale, killer, orca, grampus, sea wolf, Orcinus orca',

149: 'dugong, Dugong dugon',

150: 'sea lion',

151: 'Chihuahua',

152: 'Japanese spaniel',

153: 'Maltese dog, Maltese terrier, Maltese',

154: 'Pekinese, Pekingese, Peke',

155: 'Shih-Tzu',

156: 'Blenheim spaniel',

157: 'papillon',

158: 'toy terrier',

159: 'Rhodesian ridgeback',

160: 'Afghan hound, Afghan',

161: 'basset, basset hound',

162: 'beagle',

163: 'bloodhound, sleuthhound',

164: 'bluetick',

165: 'black-and-tan coonhound',

166: 'Walker hound, Walker foxhound',

167: 'English foxhound',

168: 'redbone',

169: 'borzoi, Russian wolfhound',

170: 'Irish wolfhound',

171: 'Italian greyhound',

172: 'whippet',

173: 'Ibizan hound, Ibizan Podenco',

174: 'Norwegian elkhound, elkhound',

175: 'otterhound, otter hound',

176: 'Saluki, gazelle hound',

177: 'Scottish deerhound, deerhound',

178: 'Weimaraner',

179: 'Staffordshire bullterrier, Staffordshire bull terrier',

180: 'American Staffordshire terrier, Staffordshire terrier, American pit bull terrier, pit bull terrier',

181: 'Bedlington terrier',

182: 'Border terrier',

183: 'Kerry blue terrier',

184: 'Irish terrier',

185: 'Norfolk terrier',

186: 'Norwich terrier',

187: 'Yorkshire terrier',

188: 'wire-haired fox terrier',

189: 'Lakeland terrier',

190: 'Sealyham terrier, Sealyham',

191: 'Airedale, Airedale terrier',

192: 'cairn, cairn terrier',

193: 'Australian terrier',

194: 'Dandie Dinmont, Dandie Dinmont terrier',

195: 'Boston bull, Boston terrier',

196: 'miniature schnauzer',

197: 'giant schnauzer',

198: 'standard schnauzer',

199: 'Scotch terrier, Scottish terrier, Scottie',

200: 'Tibetan terrier, chrysanthemum dog',

201: 'silky terrier, Sydney silky',

202: 'soft-coated wheaten terrier',

203: 'West Highland white terrier',

204: 'Lhasa, Lhasa apso',

205: 'flat-coated retriever',

206: 'curly-coated retriever',

207: 'golden retriever',

208: 'Labrador retriever',

209: 'Chesapeake Bay retriever',

210: 'German short-haired pointer',

211: 'vizsla, Hungarian pointer',

212: 'English setter',

213: 'Irish setter, red setter',

214: 'Gordon setter',

215: 'Brittany spaniel',

216: 'clumber, clumber spaniel',

217: 'English springer, English springer spaniel',

218: 'Welsh springer spaniel',

219: 'cocker spaniel, English cocker spaniel, cocker',

220: 'Sussex spaniel',

221: 'Irish water spaniel',

222: 'kuvasz',

223: 'schipperke',

224: 'groenendael',

225: 'malinois',

226: 'briard',

227: 'kelpie',

228: 'komondor',

229: 'Old English sheepdog, bobtail',

230: 'Shetland sheepdog, Shetland sheep dog, Shetland',

231: 'collie',

232: 'Border collie',

233: 'Bouvier des Flandres, Bouviers des Flandres',

234: 'Rottweiler',

235: 'German shepherd, German shepherd dog, German police dog, alsatian',

236: 'Doberman, Doberman pinscher',

237: 'miniature pinscher',

238: 'Greater Swiss Mountain dog',

239: 'Bernese mountain dog',

240: 'Appenzeller',

241: 'EntleBucher',

242: 'boxer',

243: 'bull mastiff',

244: 'Tibetan mastiff',

245: 'French bulldog',

246: 'Great Dane',

247: 'Saint Bernard, St Bernard',

248: 'Eskimo dog, husky',

249: 'malamute, malemute, Alaskan malamute',

250: 'Siberian husky',

251: 'dalmatian, coach dog, carriage dog',

252: 'affenpinscher, monkey pinscher, monkey dog',

253: 'basenji',

254: 'pug, pug-dog',

255: 'Leonberg',

256: 'Newfoundland, Newfoundland dog',

257: 'Great Pyrenees',

258: 'Samoyed, Samoyede',

259: 'Pomeranian',

260: 'chow, chow chow',

261: 'keeshond',

262: 'Brabancon griffon',

263: 'Pembroke, Pembroke Welsh corgi',

264: 'Cardigan, Cardigan Welsh corgi',

265: 'toy poodle',

266: 'miniature poodle',

267: 'standard poodle',

268: 'Mexican hairless',

269: 'timber wolf, grey wolf, gray wolf, Canis lupus',

270: 'white wolf, Arctic wolf, Canis lupus tundrarum',

271: 'red wolf, maned wolf, Canis rufus, Canis niger',

272: 'coyote, prairie wolf, brush wolf, Canis latrans',

273: 'dingo, warrigal, warragal, Canis dingo',

274: 'dhole, Cuon alpinus',

275: 'African hunting dog, hyena dog, Cape hunting dog, Lycaon pictus',

276: 'hyena, hyaena',

277: 'red fox, Vulpes vulpes',

278: 'kit fox, Vulpes macrotis',

279: 'Arctic fox, white fox, Alopex lagopus',

280: 'grey fox, gray fox, Urocyon cinereoargenteus',

281: 'tabby, tabby cat',

282: 'tiger cat',

283: 'Persian cat',

284: 'Siamese cat, Siamese',

285: 'Egyptian cat',

286: 'cougar, puma, catamount, mountain lion, painter, panther, Felis concolor',

287: 'lynx, catamount',

288: 'leopard, Panthera pardus',

289: 'snow leopard, ounce, Panthera uncia',

290: 'jaguar, panther, Panthera onca, Felis onca',

291: 'lion, king of beasts, Panthera leo',

292: 'tiger, Panthera tigris',

293: 'cheetah, chetah, Acinonyx jubatus',

294: 'brown bear, bruin, Ursus arctos',

295: 'American black bear, black bear, Ursus americanus, Euarctos americanus',

296: 'ice bear, polar bear, Ursus Maritimus, Thalarctos maritimus',

297: 'sloth bear, Melursus ursinus, Ursus ursinus',

298: 'mongoose',

299: 'meerkat, mierkat',

300: 'tiger beetle',

301: 'ladybug, ladybeetle, lady beetle, ladybird, ladybird beetle',

302: 'ground beetle, carabid beetle',

303: 'long-horned beetle, longicorn, longicorn beetle',

304: 'leaf beetle, chrysomelid',

305: 'dung beetle',

306: 'rhinoceros beetle',

307: 'weevil',

308: 'fly',

309: 'bee',

310: 'ant, emmet, pismire',

311: 'grasshopper, hopper',

312: 'cricket',

313: 'walking stick, walkingstick, stick insect',

314: 'cockroach, roach',

315: 'mantis, mantid',

316: 'cicada, cicala',

317: 'leafhopper',

318: 'lacewing, lacewing fly',

319: "dragonfly, darning needle, devil's darning needle, sewing needle, snake feeder, snake doctor, mosquito hawk, skeeter hawk",

320: 'damselfly',

321: 'admiral',

322: 'ringlet, ringlet butterfly',

323: 'monarch, monarch butterfly, milkweed butterfly, Danaus plexippus',

324: 'cabbage butterfly',

325: 'sulphur butterfly, sulfur butterfly',

326: 'lycaenid, lycaenid butterfly',

327: 'starfish, sea star',

328: 'sea urchin',

329: 'sea cucumber, holothurian',

330: 'wood rabbit, cottontail, cottontail rabbit',

331: 'hare',

332: 'Angora, Angora rabbit',

333: 'hamster',

334: 'porcupine, hedgehog',

335: 'fox squirrel, eastern fox squirrel, Sciurus niger',

336: 'marmot',

337: 'beaver',

338: 'guinea pig, Cavia cobaya',

339: 'sorrel',

340: 'zebra',

341: 'hog, pig, grunter, squealer, Sus scrofa',

342: 'wild boar, boar, Sus scrofa',

343: 'warthog',

344: 'hippopotamus, hippo, river horse, Hippopotamus amphibius',

345: 'ox',

346: 'water buffalo, water ox, Asiatic buffalo, Bubalus bubalis',

347: 'bison',

348: 'ram, tup',

349: 'bighorn, bighorn sheep, cimarron, Rocky Mountain bighorn, Rocky Mountain sheep, Ovis canadensis',

350: 'ibex, Capra ibex',

351: 'hartebeest',

352: 'impala, Aepyceros melampus',

353: 'gazelle',

354: 'Arabian camel, dromedary, Camelus dromedarius',

355: 'llama',

356: 'weasel',

357: 'mink',

358: 'polecat, fitch, foulmart, foumart, Mustela putorius',

359: 'black-footed ferret, ferret, Mustela nigripes',

360: 'otter',

361: 'skunk, polecat, wood pussy',

362: 'badger',

363: 'armadillo',

364: 'three-toed sloth, ai, Bradypus tridactylus',

365: 'orangutan, orang, orangutang, Pongo pygmaeus',

366: 'gorilla, Gorilla gorilla',

367: 'chimpanzee, chimp, Pan troglodytes',

368: 'gibbon, Hylobates lar',

369: 'siamang, Hylobates syndactylus, Symphalangus syndactylus',

370: 'guenon, guenon monkey',

371: 'patas, hussar monkey, Erythrocebus patas',

372: 'baboon',

373: 'macaque',

374: 'langur',

375: 'colobus, colobus monkey',

376: 'proboscis monkey, Nasalis larvatus',

377: 'marmoset',

378: 'capuchin, ringtail, Cebus capucinus',

379: 'howler monkey, howler',

380: 'titi, titi monkey',

381: 'spider monkey, Ateles geoffroyi',

382: 'squirrel monkey, Saimiri sciureus',

383: 'Madagascar cat, ring-tailed lemur, Lemur catta',

384: 'indri, indris, Indri indri, Indri brevicaudatus',

385: 'Indian elephant, Elephas maximus',

386: 'African elephant, Loxodonta africana',

387: 'lesser panda, red panda, panda, bear cat, cat bear, Ailurus fulgens',

388: 'giant panda, panda, panda bear, coon bear, Ailuropoda melanoleuca',

389: 'barracouta, snoek',

390: 'eel',

391: 'coho, cohoe, coho salmon, blue jack, silver salmon, Oncorhynchus kisutch',

392: 'rock beauty, Holocanthus tricolor',

393: 'anemone fish',

394: 'sturgeon',

395: 'gar, garfish, garpike, billfish, Lepisosteus osseus',

396: 'lionfish',

397: 'puffer, pufferfish, blowfish, globefish',

398: 'abacus',

399: 'abaya',

400: "academic gown, academic robe, judge's robe",

401: 'accordion, piano accordion, squeeze box',

402: 'acoustic guitar',

403: 'aircraft carrier, carrier, flattop, attack aircraft carrier',

404: 'airliner',

405: 'airship, dirigible',

406: 'altar',

407: 'ambulance',

408: 'amphibian, amphibious vehicle',

409: 'analog clock',

410: 'apiary, bee house',

411: 'apron',

412: 'ashcan, trash can, garbage can, wastebin, ash bin, ash-bin, ashbin, dustbin, trash barrel, trash bin',

413: 'assault rifle, assault gun',

414: 'backpack, back pack, knapsack, packsack, rucksack, haversack',

415: 'bakery, bakeshop, bakehouse',

416: 'balance beam, beam',

417: 'balloon',

418: 'ballpoint, ballpoint pen, ballpen, Biro',

419: 'Band Aid',

420: 'banjo',

421: 'bannister, banister, balustrade, balusters, handrail',

422: 'barbell',

423: 'barber chair',

424: 'barbershop',

425: 'barn',

426: 'barometer',

427: 'barrel, cask',

428: 'barrow, garden cart, lawn cart, wheelbarrow',

429: 'baseball',

430: 'basketball',

431: 'bassinet',

432: 'bassoon',

433: 'bathing cap, swimming cap',

434: 'bath towel',

435: 'bathtub, bathing tub, bath, tub',

436: 'beach wagon, station wagon, wagon, estate car, beach waggon, station waggon, waggon',

437: 'beacon, lighthouse, beacon light, pharos',

438: 'beaker',

439: 'bearskin, busby, shako',

440: 'beer bottle',

441: 'beer glass',

442: 'bell cote, bell cot',

443: 'bib',

444: 'bicycle-built-for-two, tandem bicycle, tandem',

445: 'bikini, two-piece',

446: 'binder, ring-binder',

447: 'binoculars, field glasses, opera glasses',

448: 'birdhouse',

449: 'boathouse',

450: 'bobsled, bobsleigh, bob',

451: 'bolo tie, bolo, bola tie, bola',

452: 'bonnet, poke bonnet',

453: 'bookcase',

454: 'bookshop, bookstore, bookstall',

455: 'bottlecap',

456: 'bow',

457: 'bow tie, bow-tie, bowtie',

458: 'brass, memorial tablet, plaque',

459: 'brassiere, bra, bandeau',

460: 'breakwater, groin, groyne, mole, bulwark, seawall, jetty',

461: 'breastplate, aegis, egis',

462: 'broom',

463: 'bucket, pail',

464: 'buckle',

465: 'bulletproof vest',

466: 'bullet train, bullet',

467: 'butcher shop, meat market',

468: 'cab, hack, taxi, taxicab',

469: 'caldron, cauldron',

470: 'candle, taper, wax light',

471: 'cannon',

472: 'canoe',

473: 'can opener, tin opener',

474: 'cardigan',

475: 'car mirror',

476: 'carousel, carrousel, merry-go-round, roundabout, whirligig',

477: "carpenter's kit, tool kit",

478: 'carton',

479: 'car wheel',

480: 'cash machine, cash dispenser, automated teller machine, automatic teller machine, automated teller, automatic teller, ATM',

481: 'cassette',

482: 'cassette player',

483: 'castle',

484: 'catamaran',

485: 'CD player',

486: 'cello, violoncello',

487: 'cellular telephone, cellular phone, cellphone, cell, mobile phone',

488: 'chain',

489: 'chainlink fence',

490: 'chain mail, ring mail, mail, chain armor, chain armour, ring armor, ring armour',

491: 'chain saw, chainsaw',

492: 'chest',

493: 'chiffonier, commode',

494: 'chime, bell, gong',

495: 'china cabinet, china closet',

496: 'Christmas stocking',

497: 'church, church building',

498: 'cinema, movie theater, movie theatre, movie house, picture palace',

499: 'cleaver, meat cleaver, chopper',

500: 'cliff dwelling',

501: 'cloak',

502: 'clog, geta, patten, sabot',

503: 'cocktail shaker',

504: 'coffee mug',

505: 'coffeepot',

506: 'coil, spiral, volute, whorl, helix',

507: 'combination lock',

508: 'computer keyboard, keypad',

509: 'confectionery, confectionary, candy store',

510: 'container ship, containership, container vessel',

511: 'convertible',

512: 'corkscrew, bottle screw',

513: 'cornet, horn, trumpet, trump',

514: 'cowboy boot',

515: 'cowboy hat, ten-gallon hat',

516: 'cradle',

517: 'crane',

518: 'crash helmet',

519: 'crate',

520: 'crib, cot',

521: 'Crock Pot',

522: 'croquet ball',

523: 'crutch',

524: 'cuirass',

525: 'dam, dike, dyke',

526: 'desk',

527: 'desktop computer',

528: 'dial telephone, dial phone',

529: 'diaper, nappy, napkin',

530: 'digital clock',

531: 'digital watch',

532: 'dining table, board',

533: 'dishrag, dishcloth',

534: 'dishwasher, dish washer, dishwashing machine',

535: 'disk brake, disc brake',

536: 'dock, dockage, docking facility',

537: 'dogsled, dog sled, dog sleigh',

538: 'dome',

539: 'doormat, welcome mat',

540: 'drilling platform, offshore rig',

541: 'drum, membranophone, tympan',

542: 'drumstick',

543: 'dumbbell',

544: 'Dutch oven',

545: 'electric fan, blower',

546: 'electric guitar',

547: 'electric locomotive',

548: 'entertainment center',

549: 'envelope',

550: 'espresso maker',

551: 'face powder',

552: 'feather boa, boa',

553: 'file, file cabinet, filing cabinet',

554: 'fireboat',

555: 'fire engine, fire truck',

556: 'fire screen, fireguard',

557: 'flagpole, flagstaff',

558: 'flute, transverse flute',

559: 'folding chair',

560: 'football helmet',

561: 'forklift',

562: 'fountain',

563: 'fountain pen',

564: 'four-poster',

565: 'freight car',

566: 'French horn, horn',

567: 'frying pan, frypan, skillet',

568: 'fur coat',

569: 'garbage truck, dustcart',

570: 'gasmask, respirator, gas helmet',

571: 'gas pump, gasoline pump, petrol pump, island dispenser',

572: 'goblet',

573: 'go-kart',

574: 'golf ball',

575: 'golfcart, golf cart',

576: 'gondola',

577: 'gong, tam-tam',

578: 'gown',

579: 'grand piano, grand',

580: 'greenhouse, nursery, glasshouse',

581: 'grille, radiator grille',

582: 'grocery store, grocery, food market, market',

583: 'guillotine',

584: 'hair slide',

585: 'hair spray',

586: 'half track',

587: 'hammer',

588: 'hamper',

589: 'hand blower, blow dryer, blow drier, hair dryer, hair drier',

590: 'hand-held computer, hand-held microcomputer',

591: 'handkerchief, hankie, hanky, hankey',

592: 'hard disc, hard disk, fixed disk',

593: 'harmonica, mouth organ, harp, mouth harp',

594: 'harp',

595: 'harvester, reaper',

596: 'hatchet',

597: 'holster',

598: 'home theater, home theatre',

599: 'honeycomb',

600: 'hook, claw',

601: 'hoopskirt, crinoline',

602: 'horizontal bar, high bar',

603: 'horse cart, horse-cart',

604: 'hourglass',

605: 'iPod',

606: 'iron, smoothing iron',

607: "jack-o'-lantern",

608: 'jean, blue jean, denim',

609: 'jeep, landrover',

610: 'jersey, T-shirt, tee shirt',

611: 'jigsaw puzzle',

612: 'jinrikisha, ricksha, rickshaw',

613: 'joystick',

614: 'kimono',

615: 'knee pad',

616: 'knot',

617: 'lab coat, laboratory coat',

618: 'ladle',

619: 'lampshade, lamp shade',

620: 'laptop, laptop computer',

621: 'lawn mower, mower',

622: 'lens cap, lens cover',

623: 'letter opener, paper knife, paperknife',

624: 'library',

625: 'lifeboat',

626: 'lighter, light, igniter, ignitor',

627: 'limousine, limo',

628: 'liner, ocean liner',

629: 'lipstick, lip rouge',

630: 'Loafer',

631: 'lotion',

632: 'loudspeaker, speaker, speaker unit, loudspeaker system, speaker system',

633: "loupe, jeweler's loupe",

634: 'lumbermill, sawmill',

635: 'magnetic compass',

636: 'mailbag, postbag',

637: 'mailbox, letter box',

638: 'maillot',

639: 'maillot, tank suit',

640: 'manhole cover',

641: 'maraca',

642: 'marimba, xylophone',

643: 'mask',

644: 'matchstick',

645: 'maypole',

646: 'maze, labyrinth',

647: 'measuring cup',

648: 'medicine chest, medicine cabinet',

649: 'megalith, megalithic structure',

650: 'microphone, mike',

651: 'microwave, microwave oven',

652: 'military uniform',

653: 'milk can',

654: 'minibus',

655: 'miniskirt, mini',

656: 'minivan',

657: 'missile',

658: 'mitten',

659: 'mixing bowl',

660: 'mobile home, manufactured home',

661: 'Model T',

662: 'modem',

663: 'monastery',

664: 'monitor',

665: 'moped',

666: 'mortar',

667: 'mortarboard',

668: 'mosque',

669: 'mosquito net',

670: 'motor scooter, scooter',

671: 'mountain bike, all-terrain bike, off-roader',

672: 'mountain tent',

673: 'mouse, computer mouse',

674: 'mousetrap',

675: 'moving van',

676: 'muzzle',

677: 'nail',

678: 'neck brace',

679: 'necklace',

680: 'nipple',

681: 'notebook, notebook computer',

682: 'obelisk',

683: 'oboe, hautboy, hautbois',

684: 'ocarina, sweet potato',

685: 'odometer, hodometer, mileometer, milometer',

686: 'oil filter',

687: 'organ, pipe organ',

688: 'oscilloscope, scope, cathode-ray oscilloscope, CRO',

689: 'overskirt',

690: 'oxcart',

691: 'oxygen mask',

692: 'packet',

693: 'paddle, boat paddle',

694: 'paddlewheel, paddle wheel',

695: 'padlock',

696: 'paintbrush',

697: "pajama, pyjama, pj's, jammies",

698: 'palace',

699: 'panpipe, pandean pipe, syrinx',

700: 'paper towel',

701: 'parachute, chute',

702: 'parallel bars, bars',

703: 'park bench',

704: 'parking meter',

705: 'passenger car, coach, carriage',

706: 'patio, terrace',

707: 'pay-phone, pay-station',

708: 'pedestal, plinth, footstall',

709: 'pencil box, pencil case',

710: 'pencil sharpener',

711: 'perfume, essence',

712: 'Petri dish',

713: 'photocopier',

714: 'pick, plectrum, plectron',

715: 'pickelhaube',

716: 'picket fence, paling',

717: 'pickup, pickup truck',

718: 'pier',

719: 'piggy bank, penny bank',

720: 'pill bottle',

721: 'pillow',

722: 'ping-pong ball',

723: 'pinwheel',

724: 'pirate, pirate ship',

725: 'pitcher, ewer',

726: "plane, carpenter's plane, woodworking plane",

727: 'planetarium',

728: 'plastic bag',

729: 'plate rack',

730: 'plow, plough',

731: "plunger, plumber's helper",

732: 'Polaroid camera, Polaroid Land camera',

733: 'pole',

734: 'police van, police wagon, paddy wagon, patrol wagon, wagon, black Maria',

735: 'poncho',

736: 'pool table, billiard table, snooker table',

737: 'pop bottle, soda bottle',

738: 'pot, flowerpot',

739: "potter's wheel",

740: 'power drill',

741: 'prayer rug, prayer mat',

742: 'printer',

743: 'prison, prison house',

744: 'projectile, missile',

745: 'projector',

746: 'puck, hockey puck',

747: 'punching bag, punch bag, punching ball, punchball',

748: 'purse',

749: 'quill, quill pen',

750: 'quilt, comforter, comfort, puff',

751: 'racer, race car, racing car',

752: 'racket, racquet',

753: 'radiator',

754: 'radio, wireless',

755: 'radio telescope, radio reflector',

756: 'rain barrel',

757: 'recreational vehicle, RV, R.V.',

758: 'reel',

759: 'reflex camera',

760: 'refrigerator, icebox',

761: 'remote control, remote',

762: 'restaurant, eating house, eating place, eatery',

763: 'revolver, six-gun, six-shooter',

764: 'rifle',

765: 'rocking chair, rocker',

766: 'rotisserie',

767: 'rubber eraser, rubber, pencil eraser',

768: 'rugby ball',

769: 'rule, ruler',

770: 'running shoe',

771: 'safe',

772: 'safety pin',

773: 'saltshaker, salt shaker',

774: 'sandal',

775: 'sarong',

776: 'sax, saxophone',

777: 'scabbard',

778: 'scale, weighing machine',

779: 'school bus',

780: 'schooner',

781: 'scoreboard',

782: 'screen, CRT screen',

783: 'screw',

784: 'screwdriver',

785: 'seat belt, seatbelt',

786: 'sewing machine',

787: 'shield, buckler',

788: 'shoe shop, shoe-shop, shoe store',

789: 'shoji',

790: 'shopping basket',

791: 'shopping cart',

792: 'shovel',

793: 'shower cap',

794: 'shower curtain',

795: 'ski',

796: 'ski mask',

797: 'sleeping bag',

798: 'slide rule, slipstick',

799: 'sliding door',

800: 'slot, one-armed bandit',

801: 'snorkel',

802: 'snowmobile',

803: 'snowplow, snowplough',

804: 'soap dispenser',

805: 'soccer ball',

806: 'sock',

807: 'solar dish, solar collector, solar furnace',

808: 'sombrero',

809: 'soup bowl',

810: 'space bar',

811: 'space heater',

812: 'space shuttle',

813: 'spatula',

814: 'speedboat',

815: "spider web, spider's web",

816: 'spindle',

817: 'sports car, sport car',

818: 'spotlight, spot',

819: 'stage',

820: 'steam locomotive',

821: 'steel arch bridge',

822: 'steel drum',

823: 'stethoscope',

824: 'stole',

825: 'stone wall',

826: 'stopwatch, stop watch',

827: 'stove',

828: 'strainer',

829: 'streetcar, tram, tramcar, trolley, trolley car',

830: 'stretcher',

831: 'studio couch, day bed',

832: 'stupa, tope',

833: 'submarine, pigboat, sub, U-boat',

834: 'suit, suit of clothes',

835: 'sundial',

836: 'sunglass',

837: 'sunglasses, dark glasses, shades',

838: 'sunscreen, sunblock, sun blocker',

839: 'suspension bridge',

840: 'swab, swob, mop',

841: 'sweatshirt',

842: 'swimming trunks, bathing trunks',

843: 'swing',

844: 'switch, electric switch, electrical switch',

845: 'syringe',

846: 'table lamp',

847: 'tank, army tank, armored combat vehicle, armoured combat vehicle',

848: 'tape player',

849: 'teapot',

850: 'teddy, teddy bear',

851: 'television, television system',

852: 'tennis ball',

853: 'thatch, thatched roof',

854: 'theater curtain, theatre curtain',

855: 'thimble',

856: 'thresher, thrasher, threshing machine',

857: 'throne',

858: 'tile roof',

859: 'toaster',

860: 'tobacco shop, tobacconist shop, tobacconist',

861: 'toilet seat',

862: 'torch',

863: 'totem pole',

864: 'tow truck, tow car, wrecker',

865: 'toyshop',

866: 'tractor',

867: 'trailer truck, tractor trailer, trucking rig, rig, articulated lorry, semi',

868: 'tray',

869: 'trench coat',

870: 'tricycle, trike, velocipede',

871: 'trimaran',

872: 'tripod',

873: 'triumphal arch',

874: 'trolleybus, trolley coach, trackless trolley',

875: 'trombone',

876: 'tub, vat',

877: 'turnstile',

878: 'typewriter keyboard',

879: 'umbrella',

880: 'unicycle, monocycle',

881: 'upright, upright piano',

882: 'vacuum, vacuum cleaner',

883: 'vase',

884: 'vault',

885: 'velvet',

886: 'vending machine',

887: 'vestment',

888: 'viaduct',

889: 'violin, fiddle',

890: 'volleyball',

891: 'waffle iron',

892: 'wall clock',

893: 'wallet, billfold, notecase, pocketbook',

894: 'wardrobe, closet, press',

895: 'warplane, military plane',

896: 'washbasin, handbasin, washbowl, lavabo, wash-hand basin',

897: 'washer, automatic washer, washing machine',

898: 'water bottle',

899: 'water jug',

900: 'water tower',

901: 'whiskey jug',

902: 'whistle',

903: 'wig',

904: 'window screen',

905: 'window shade',

906: 'Windsor tie',

907: 'wine bottle',

908: 'wing',

909: 'wok',

910: 'wooden spoon',

911: 'wool, woolen, woollen',

912: 'worm fence, snake fence, snake-rail fence, Virginia fence',

913: 'wreck',

914: 'yawl',

915: 'yurt',

916: 'web site, website, internet site, site',

917: 'comic book',

918: 'crossword puzzle, crossword',

919: 'street sign',

920: 'traffic light, traffic signal, stoplight',

921: 'book jacket, dust cover, dust jacket, dust wrapper',

922: 'menu',

923: 'plate',

924: 'guacamole',

925: 'consomme',

926: 'hot pot, hotpot',

927: 'trifle',

928: 'ice cream, icecream',

929: 'ice lolly, lolly, lollipop, popsicle',

930: 'French loaf',

931: 'bagel, beigel',

932: 'pretzel',

933: 'cheeseburger',

934: 'hotdog, hot dog, red hot',

935: 'mashed potato',

936: 'head cabbage',

937: 'broccoli',

938: 'cauliflower',

939: 'zucchini, courgette',

940: 'spaghetti squash',

941: 'acorn squash',

942: 'butternut squash',

943: 'cucumber, cuke',

944: 'artichoke, globe artichoke',

945: 'bell pepper',

946: 'cardoon',

947: 'mushroom',

948: 'Granny Smith',

949: 'strawberry',

950: 'orange',

951: 'lemon',

952: 'fig',

953: 'pineapple, ananas',

954: 'banana',

955: 'jackfruit, jak, jack',

956: 'custard apple',

957: 'pomegranate',

958: 'hay',

959: 'carbonara',

960: 'chocolate sauce, chocolate syrup',

961: 'dough',

962: 'meat loaf, meatloaf',

963: 'pizza, pizza pie',

964: 'potpie',

965: 'burrito',

966: 'red wine',

967: 'espresso',

968: 'cup',

969: 'eggnog',

970: 'alp',

971: 'bubble',

972: 'cliff, drop, drop-off',

973: 'coral reef',

974: 'geyser',

975: 'lakeside, lakeshore',

976: 'promontory, headland, head, foreland',

977: 'sandbar, sand bar',

978: 'seashore, coast, seacoast, sea-coast',

979: 'valley, vale',

980: 'volcano',

981: 'ballplayer, baseball player',

982: 'groom, bridegroom',

983: 'scuba diver',

984: 'rapeseed',

985: 'daisy',

986: "yellow lady's slipper, yellow lady-slipper, Cypripedium calceolus, Cypripedium parviflorum",

987: 'corn',

988: 'acorn',

989: 'hip, rose hip, rosehip',

990: 'buckeye, horse chestnut, conker',

991: 'coral fungus',

992: 'agaric',

993: 'gyromitra',

994: 'stinkhorn, carrion fungus',

995: 'earthstar',

996: 'hen-of-the-woods, hen of the woods, Polyporus frondosus, Grifola frondosa',

997: 'bolete',

998: 'ear, spike, capitulum',

999: 'toilet tissue, toilet paper, bathroom tissue'}

print imagenet\_labels[label]

count = 0

for root, dirs, files in os.walk("val/images\_new"):

path = root.split(os.sep)

for file in files:

img = imresize(imread("val/images\_new/"+file, mode='RGB'), (224, 224)).astype(np.float32)

img[:, :, 0] -= 123.68

img[:, :, 1] -= 116.779

img[:, :, 2] -= 103.939

img[:,:,[0,1,2]] = img[:,:,[2,1,0]]

img = img.transpose((2, 0, 1))

img = np.expand\_dims(img, axis=0)

out = model.predict(img) # note: the model has three outputs

predicted\_label = np.argmax(out[2])

#print(labels\_list[file])

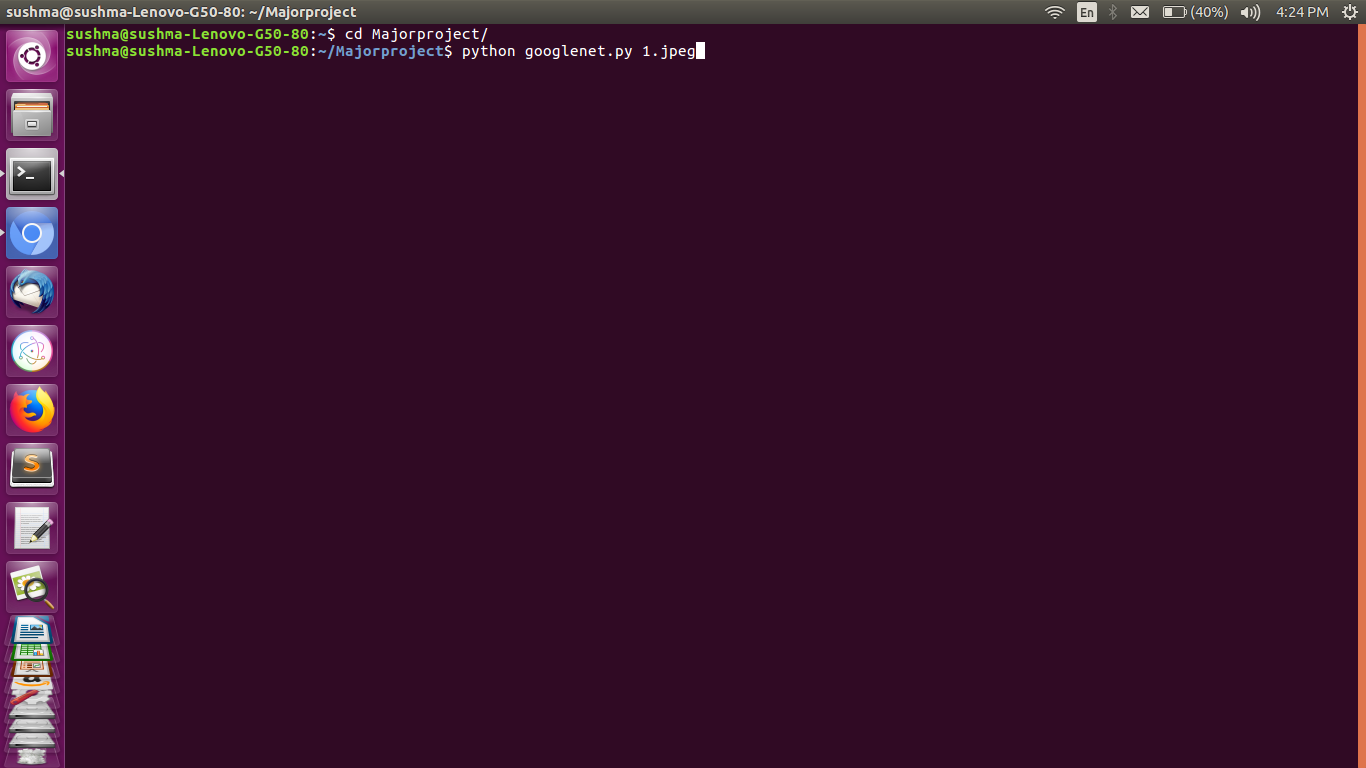
actual\_label = final\_labels[labels\_list[file]

if(predicted\_label==actual\_label):

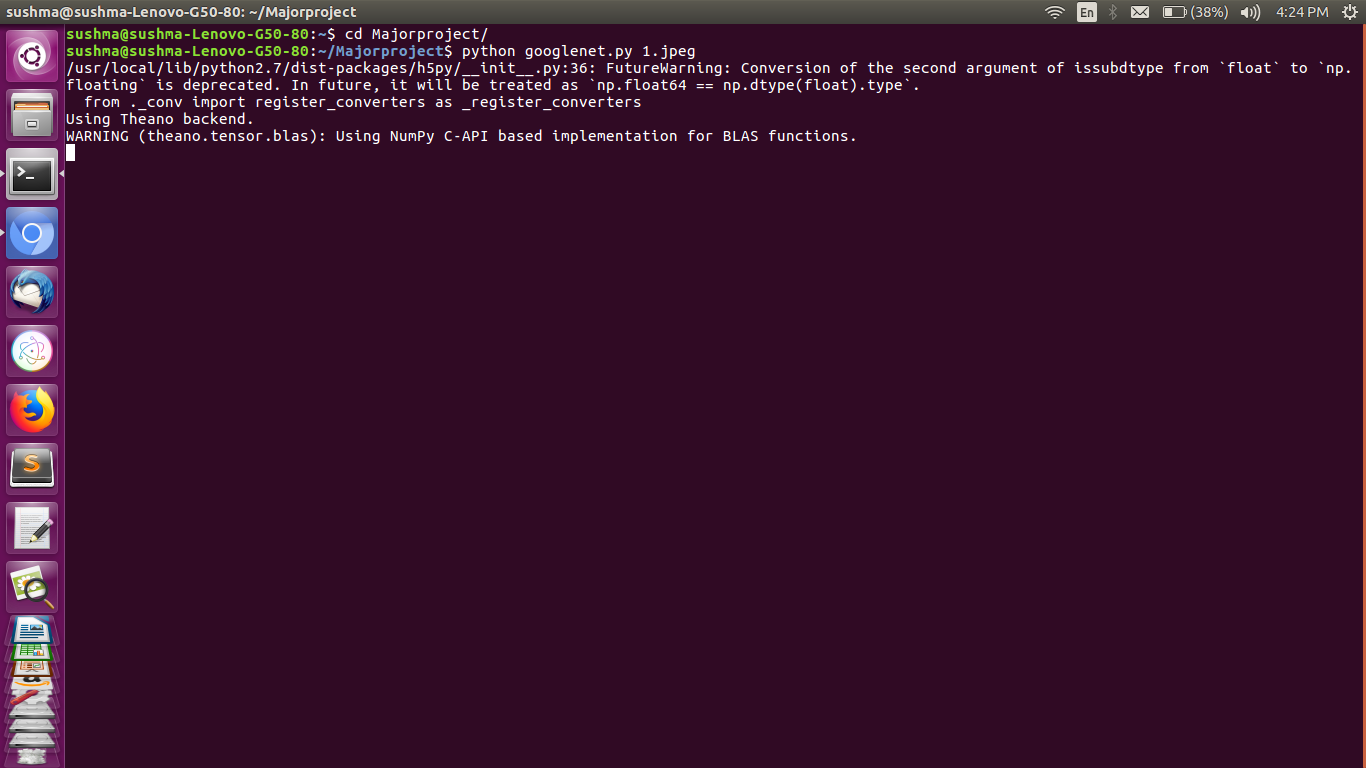
count=count+1

print((count/54)\*100)

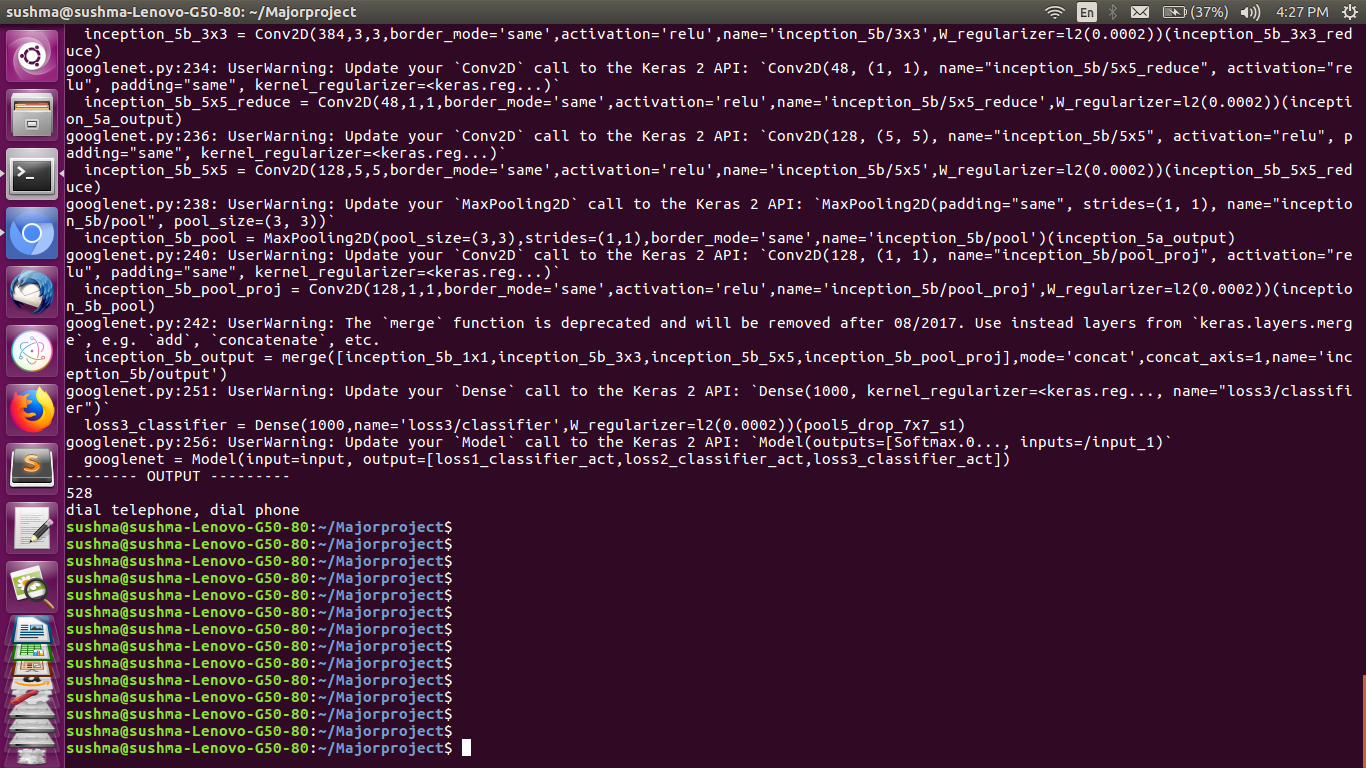
**4.2 Output Screens**

****

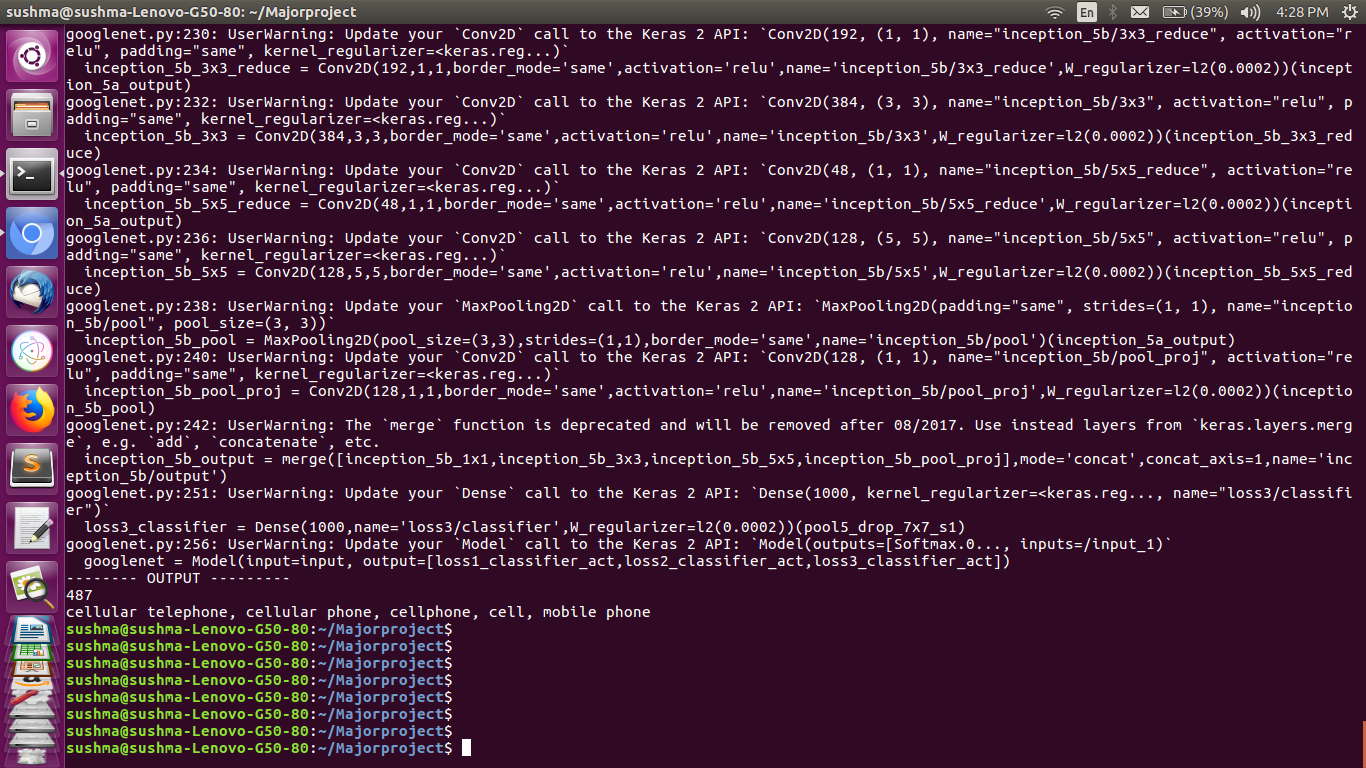
**Figure 4.1 : Running the project file**



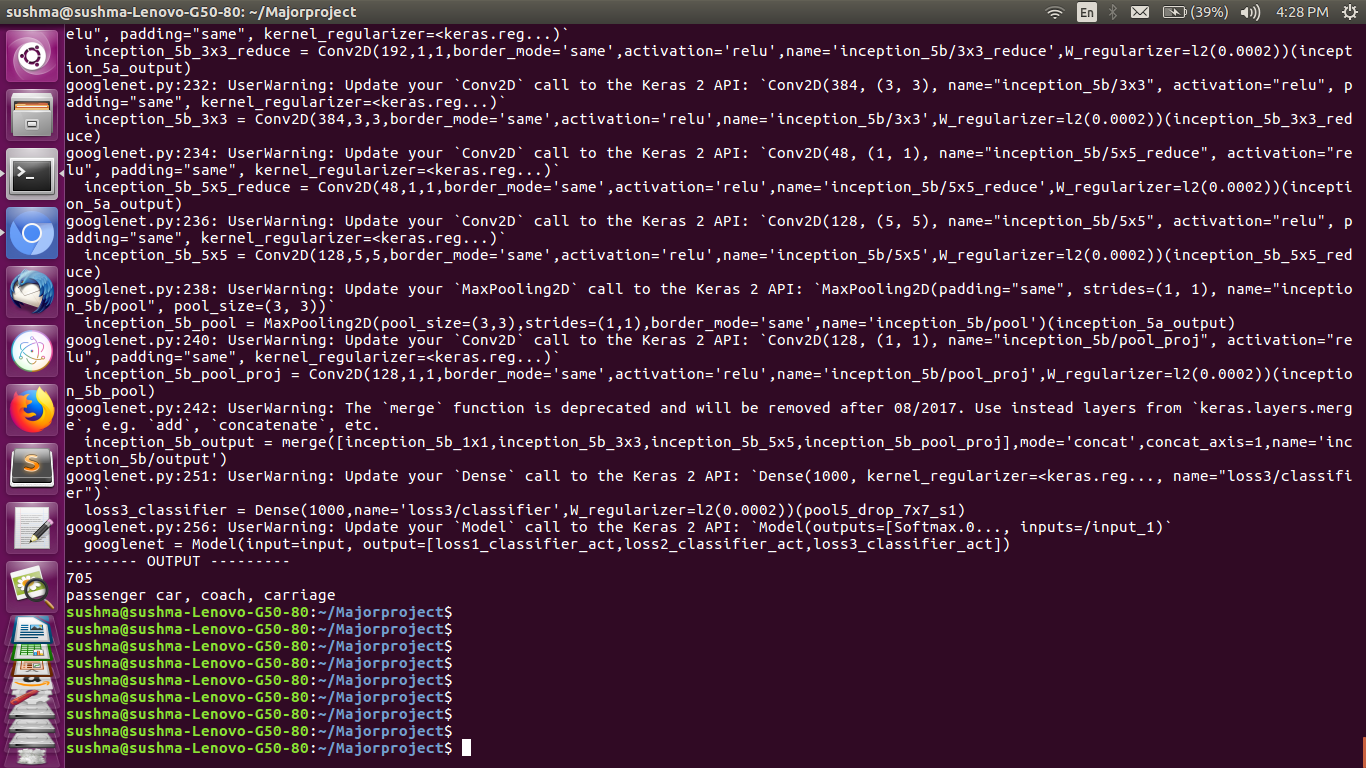
**Figure 4.2 : Running on Theano as Backend**

****

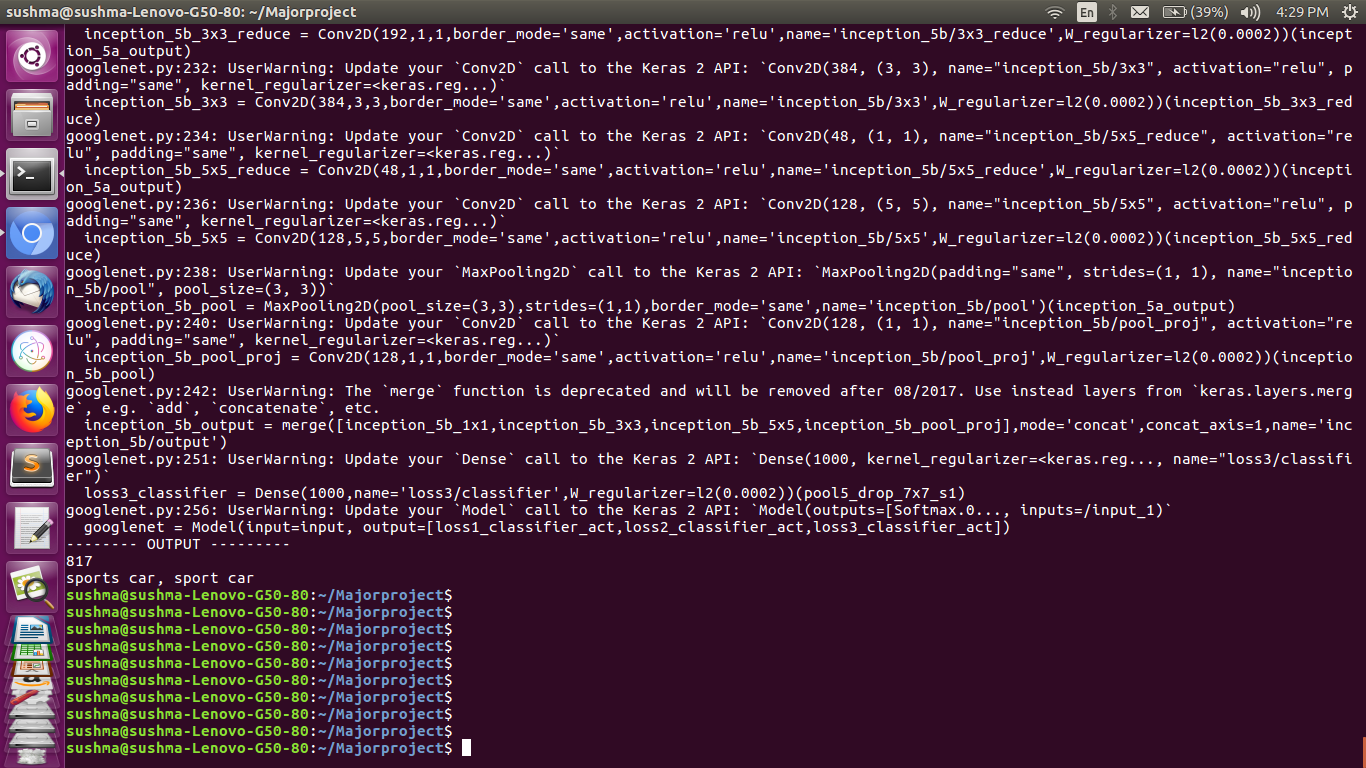
**Figure 4.3 : Output-I**



**Figure 4.4 : Output-II**

****

**Figure 4.5 : Output-III**



**Figure 4.6 : Output-IV**

**5. Test Cases**

Software testing is the process used to help identify the correctness, completeness, security, and quality of developed computer software. Testing is a process of technical investigation, performed on behalf of stakeholders, that is intended to reveal quality related information about the product with respect to the context in which it is intended to operate.

**Unit Testing**

Unit testing, a testing technique using which individual modules are tested to determine if there are any issues by the developer himself/herself. It is concerned with functional correctness of standalone modules. Unit testing is a component of test driven development (TTD), a pragmatic methodology that takes a meticulous approach to building a product by means of continual testing and revision. Test driven development requires that developers first write failing unit tests. Then, they write code and re factor the application until the test passes. TTD typically results in an explicit and predictable code base.

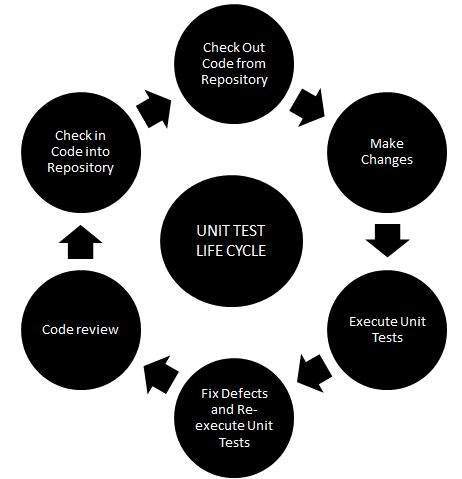
Unit testing involves only those characteristics that are vital to the performance of the unit under test. This encourages developers to modify the source code without immediate concerns about how such changes might affect the functioning of other units or the program as a whole. Once all of the units in a program have been found to be working in the most efficient and error free manner possible, larger components of the program can be evaluated by means of integration testing.

Unit testing does have steep learning curve. The development team needs to learn what unit testing is, how to unit test, what to unit test and how to use automated software tools to facilitate the process on an ongoing basis. The great benefit to unit testing is that the earlier the problem is identified, the fewer compound errors occur. A compound error is one that doesn’t seem to break anything at first, but eventually conflicts with something down the line and results in a problem.

The main aim is to isolate each unit of the system to identify, analyze and fix the defects.

Advantages :

* Reduces the defects in the newly developed features or reduces the bugs when changing the existing functionality.
* Reduces cost of testing as defects are captured in very early phase.
* Improves design and allows better refactoring of code.
* Unit tests, when integrated with build gives the quality of build as well.



**Figure 5.1 : Unit Testing life cycle**

**5.1 Unit Test Cases**

The GoogLeNet has 1000 class labels as different image categories. To test on 1000 labels cannot give us satisfied results, hence we opted for ‘Electronic Goods and Vehicular objects’ as main class labels to be focused on to test the accuracy of our modeled code.

The basic testing that is unit testing is done over the model by passing images as arguments over the command line and checking the results with expected outputs. The results were so pleasing that there was not much differences between predicted and expected outputs. As a result of observation, we finally simulated that the ‘GoogLeNet’ model efficiently outputs the image’s deep features along with its recognition.

The following are few test cases where we gave input as an image. Here, image’s type is not restricted, the type can be of ‘JPG, PNG or JPEG’ and the resolution and size of image also doesn’t matter because once the image is inputted, it will be resized to desired width and height in order to maintain consistency among different layers of convolutions.

**Table 4 : Unit test cases - I**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.No** | **Input** | **Format** | **Expected Outcome** | **Actual Outcome** |
| 1. |  | JPEG | bus, Passenger double-decker bus | Passenger car, coach, carriage |
| 2. |  | JPG | mobile phone, cellular device | cellular telephone, cellular phone, cellphone, cell, mobile phone |

**Table 5 : Unit test cases - II**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.No** | **Input** | **Format** | **Expected Outcome** | **Actual Outcome** |
| 3. |  | PNG | motor bike | motor scooter, scooter, bike |
| 4. |  | JPEG | telephone, dial phone | dial telephone, dial phone |

**6. Conclusion and Future enhancements**

The image recognition is done using Inception model, where the image is passed over layers of convolutional neural nets and also been applied deeper filters to view the deepest features of the image. An input image is taken and on filters like 1x1, 3x3, 5x5 the deep feature extraction is done. The output displayed is based on the class label of which input image belong to. Hence, the output of the simulation is image’s class type out of 1000 classes which are pre-trained in GoogLeNet, along with the deep features.

The project has led us exposure to industry culture. We made progress through “scrum meets”. It relates of good experience working in team. If there’s an error then we applied “Dive Deep” concept to find root cause.

**Scope for Future enhancement**

Scope of study can be enlarged by considering more complex images i.e, image with multiple objects. This code can be integrated with Arduino microprocessor for real time applications like training robots for image recognition. In our project we only simulated the results of a research paper and inception model is implemented for extraction of deeper features. There are many statistical functions available in Keras API, so different image processing effects can be applied to the image for testing and working of those functions. Our simulations are basis of ground truth that pre-trained set of 1000 classes of images i.e, GoogLeNet has highest accuracy in recognition of images. Further the layers of inception model can be increased to attain highest level of accuracy which can be done by changing the GoogLeNet architecture.

**7. References**

[1] Szegedy Christian, Wei Liu, Published in: Computer Vision and Pattern Recognition (CVPR), 2015 IEEE Conference, Available: http://ieeexplore.ieee.org/document/7298594/

[2] Research at Google. Available: https://research.google.com/pubs/pub43022.html

[3] Know your meme: We need to go deeper http://knowyourmeme.com/memes/we-need-to-go-deeper

[4] Going Deeper with convolutions : https://www.cs.unc.edu/~wliu/papers/GoogLeNet.pdf

[5] S. Arora, A. Bhaskara, R. Ge, and T. Ma. Provable bounds for learning some deep representations. CoRR, abs/1310.6343, 2013

[6] D. Erhan, C. Szegedy, A. Toshev, and D. Anguelov. Scalable object detection using deep neural networks. In CVPR, 2014

[7] F. Song and J. Dongarra. Scaling up matrix computations on shared-memory many core systems with 1000 cpu cores. In Proceedings of the 28th ACM International Conference on Supercomputing, ICS ’14, pages 333–342, New York, NY, USA, 2014

[8] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In Computer Vision and Pattern Recognition, 2014. CVPR 2014. IEEE Conference on, 2014