# Image recognition - Identify rooms of the house using Neural Network

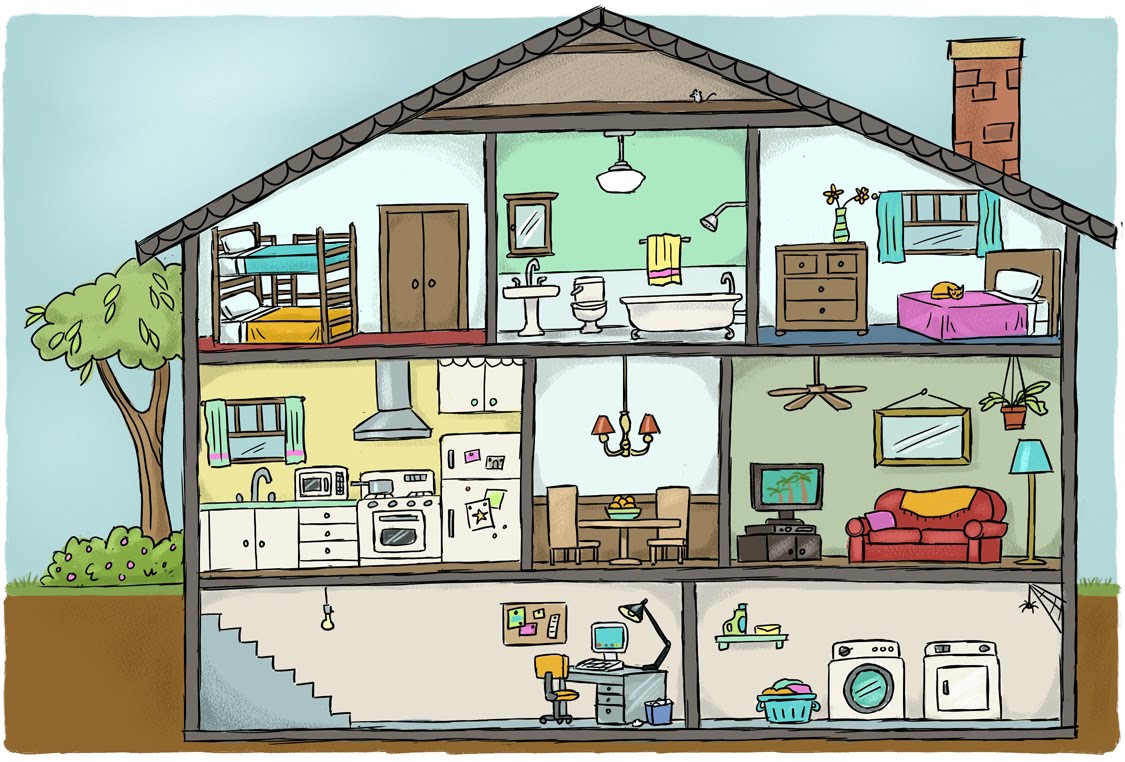


Image recognition, in the context of machine vision, is the ability of software to [identify objects](https://whatis.techtarget.com/definition/object-recognition), places, people, writing and actions in images. Computers can use machine vision technologies in combination with a camera and [artificial intelligence](https://searchenterpriseai.techtarget.com/definition/AI-Artificial-Intelligence) software to achieve image recognition.

Image recognition is used to perform a large number of machine-based visual tasks, such as labeling the content of images with [meta-tags](https://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](https://searchenterpriseai.techtarget.com/definition/deep-learning-deep-neural-network). Performance is best on convolutional [neural net processors](https://searchenterpriseai.techtarget.com/definition/neural-network) as the specific task otherwise requires massive amounts of power for its compute-intensive nature. Image recognition [algorithms](https://whatis.techtarget.com/definition/algorithm) can function by use of comparative [3D models](https://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.

Current and future applications of image recognition include smart photo libraries, targeted advertising, the interactivity of media, accessibility for the visually impaired and enhanced research capabilities. [Google](https://searchcio.techtarget.com/definition/Google-The-Company), [Facebook](https://whatis.techtarget.com/definition/Facebook), [Microsoft](https://searchwindowsserver.techtarget.com/definition/Microsoft), [Apple](https://whatis.techtarget.com/definition/Apple) and [Pinterest](https://whatis.techtarget.com/definition/Pinterest) are among the many companies that are investing significant resources and research into image recognition and related applications. Privacy concerns over image recognition and similar technologies are controversial as these companies can pull a large volume of data from user photos uploaded to their social media platforms.

In the analysis below neural networks and deep learning algorithms will be used to identify which room of a household the image is depicting.

One use case there the final algorithm can be used in is for smart vacuum cleaners, where they will have to perform different cleaning processes depending on the room of the house they are at. In general there could potentially be numerous application with the rise of IoT.

# Data collection – Web scraping



To **gather the images google image** search engine was scraped for specific search queries like ‘kitchen’, ‘old kitchen’, ‘modern kitchen’, ‘bathroom’ etc

More specifically the steps taken to gather our labeled images were the following:

* Logging output for after code runs

logging.basicConfig(filename='myapp.log', level=logging.INFO)

agent\_version = '%.2f' % (random.randint(20, 100) + random.randint(1, 100)/float(100))

* Create a header for the get request when downloading the images

headers = {

'User-Agent': 'Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36',

'User-Agent': 'Mozilla/5.0 (compatible; MSIE 8\.0; Windows NT 5\.1; SV1) Chrome/%s.2924.87 Safari/537.36' % agent\_version}

os = 'linux'

* Open a browser but not visible

display = Display(visible=0, size=(800, 800))

display.start()display = Display(visible=0, size=(800, 800))

display.start()

* Google url for image searches

basic\_url = 'https://www.google.gr/search?q={}&source=lnms&tbm=isch&sa=X&ved=0ahUKEwjT-Kj668vbAhUD\_iwKHSlVAKwQ\_AUICigB&biw=1280&bih=615'

* For every search term include search term in google url

url = basic\_url.format(str(param))

* Run chrome driver for selenium to start the scraping process

driver = webdriver.Chrome("/usr/lib/chromium-browser/chromedriver")

* Give time for selenium to start

wait = WebDriverWait(driver, 100)

* Perform get request for the google url

driver.get(str(url))

* Sometimes google opens a bot page first wait until class name 'med'. It waits up to 100 seconds

wait.until(lambda driver: driver.find\_elements(By.CLASS\_NAME, 'med'))

* Scroll down to reveal more pictures

for scroll in range(10):

driver.execute\_script("window.scrollBy(0,1000000)")

time.sleep(0.2)

time.sleep(0.5)

* List to store the image urls after scrapping the google results

img\_urls = []

* Find all photos, this is where in the html from google we can find the images

images = driver.find\_elements\_by\_xpath("//div[@class='rg\_meta notranslate']")

print('---- Number of pictures found: ', len(images))

for image in images:

get image information from every html code chunk. the information i am retrieving is the image url

img\_url = json.loads(image.get\_attribute("innerHTML"))["ou"]

* Create an id for the image url we retrieved, this is the name i am going to give to the image

img\_id = str(uuid.uuid4())

* Store image url and image name in a dictionary

img\_dict = {'img\_url':img\_url, 'img\_id':img\_id}

* Append the above information for every loop i.e. for every image

img\_urls.append(img\_dict)

* Close selenium

We only use selenium to get the information from google image search. once we have the image urls we just do a simple get request. No need for selenium driver.

driver.quit()

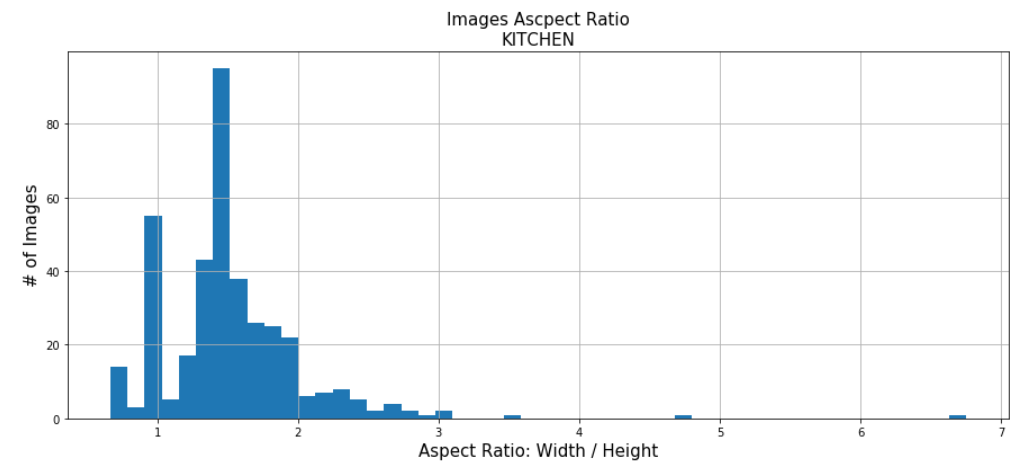
* Create dir if it does not exist
* For every search term create the respective path file in order to store the
* Download the pictures, for every image url

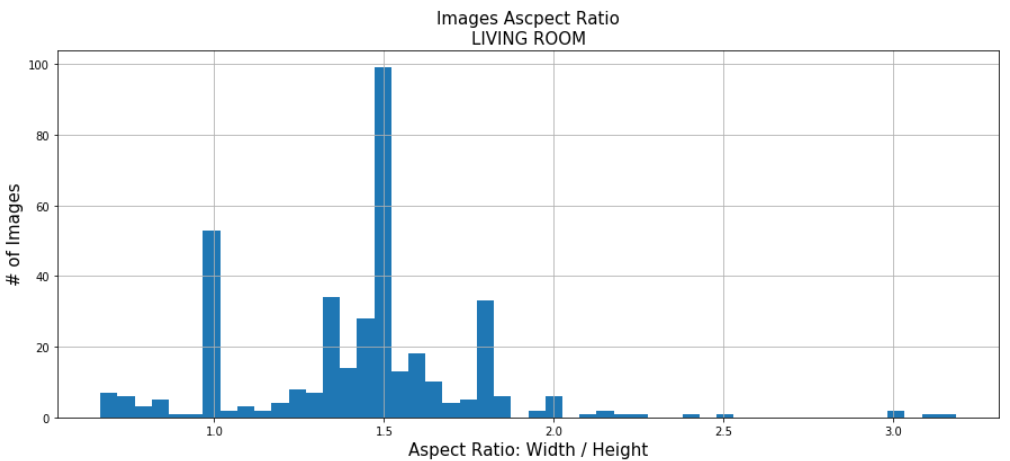
# Data Processing

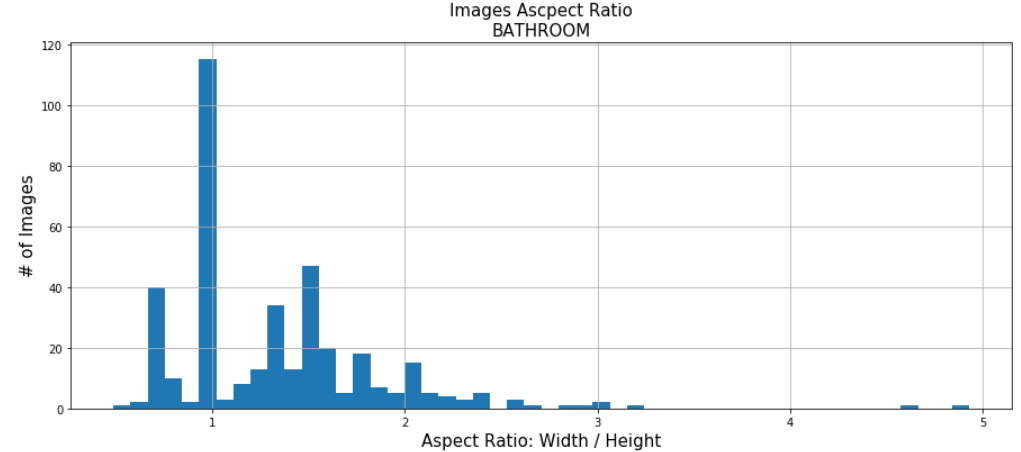
The biggest challenge in terms of data wrangling is to **resize** the images so as all image to be the same, in terms of size and **reshape** them in a form that can be processed.

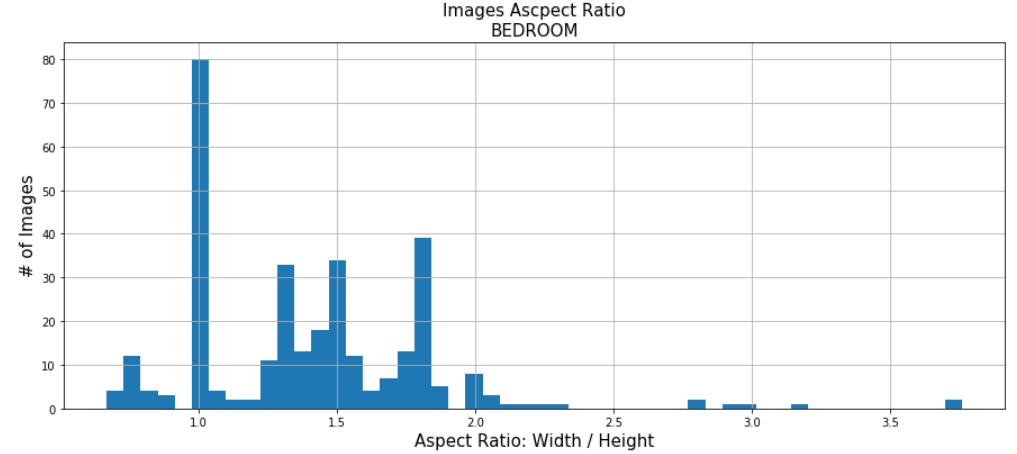
Resizing:

As we can see from the charts above the size of the pictures varies a lot. In order to proceed with the analysis of the images have to have the same size. So all photos have been resized to 100x100.









Before the resizing we need to get image sizes and paths

sizes = defaultdict(list)

imgs = defaultdict(list)

for path in paths:

pics = os.listdir(path)

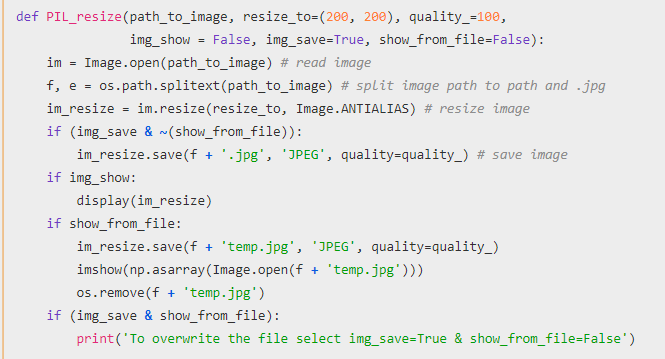
for pic in pics:

try:

sizes[path.split('/')[-2]].append(Image.open(path+pic).size)

imgs[path.split('/')[-2]].append(path+pic)

We have created a custom function to resize the images:



Then the resizing prosses can follow. The pictures below are an example of a picture before and after the resizing.

imshow(np.asarray(Image.open(imgs['livingroom'][10])));



PIL\_resize(path\_to\_image=imgs['livingroom'][120], resize\_to=(100,100), quality\_=100, img\_show=False, img\_save=False, show\_from\_file=True)



Reshaping

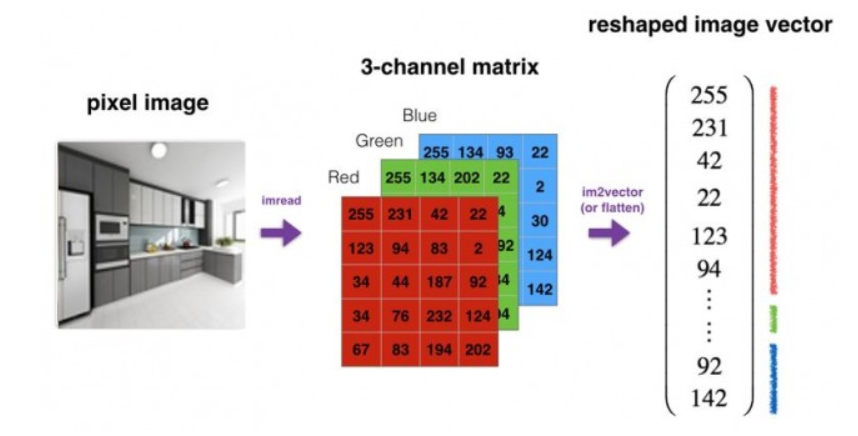
Image to vector conversion: Since we have chosen the 100x100 size we have 30000 pixels / RGB per image.

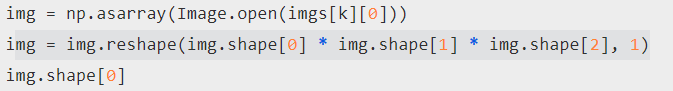
img = np.asarray(Image.open(imgs[k][0]))

img = img.reshape(img.shape[0] \* img.shape[1] \* img.shape[2], 1)

img.shape[0]

Reshaping:





We have 3000 pixels / RGB per image

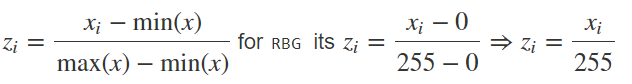
We create an initial matrix X of dimantions n, m where n is the number of pixels \* rgb colours and m are the number of images

X = np.zeros((img.shape[0], imgs\_count))

The labeled data of the image are stored in list Y

Y = list()

For every image we have stored we create respective X and Y we open the images convert the image to a vector of size x\_pixels \* y\_pixels \* 3, 1 and store it in the respective column of matrix X, for example the first image in going to be stored in the first column of X. We normalize using the nim-mx method.



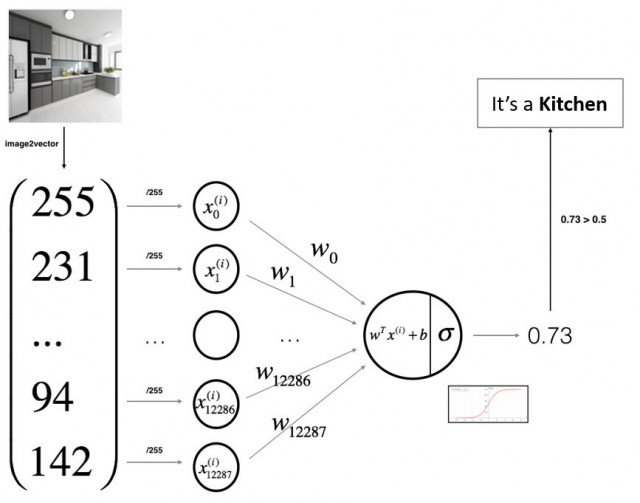
X[:, i] = img.reshape(img.shape[0] \* img.shape[1] \* img.shape[2]) / 255

We then append the folder name in which the image was saved.

# Predictive Models

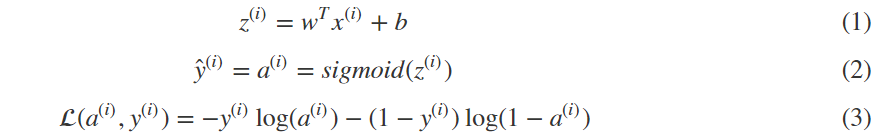
# Neural Network with no hidden layer - Logistic Regression

You will first test a **Logistic Regression** activation function, using a Neural Network mindset.



#### **Mathematical expression of the algorithm**:

For one example x(i):



The third equation above is the logistic regression loss function. Loss function would be the square root error: L(y',y) = 1/2 (y' - y)^2. This notation will not be used though because it leads us to optimization problem which is non convex, means it contains local optimum points.

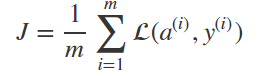
The function that we will use is : L(a,y) = - (y\*log(a) + (1-y)\*log(1-a))

To explain the last function lets see:

If y = 1==> L(a,1) = -log(a)==> we want a to be the largest ==>a biggest value is 1

If y = 0==> L(a,0) = -log(1-a)==> we want 1-a to be the largest ==>a to be smaller as possible because it can only has 1 value.

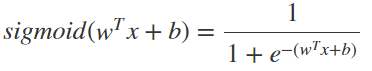
The cost is then computed by summing over all training examples (The loss function computes the error for a single training example; the cost function is the average of the loss functions of the entire training set):

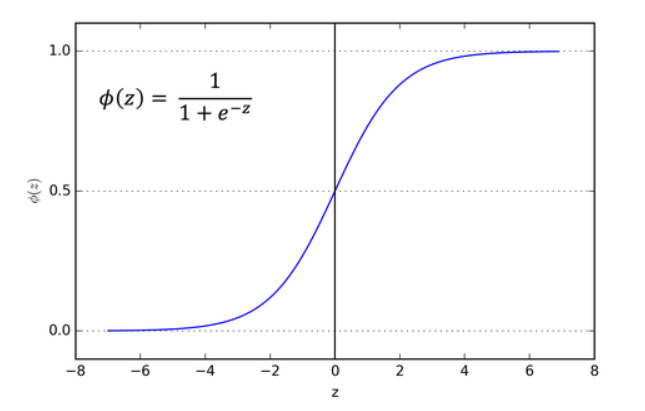


#### Helper functions – Sigmoid

The role of an activation function is to introduce nonlinearity. An advantage of this is that the output is mapped from a range of 0 and 1, making it easier to alter weights in the future.

There are many activation functions out there. In this case, we'll stick to one of the more popular ones - the sigmoid function.





Key steps*:*

Initialize the parameters of the model

We want to predict the w and p parameters that minimize the cost function. First we initialize the parameters to 0,0 or initialize them to a random value in the convex function and then try to improve the values the reach minimum value. In Logistic regression 0,0 is used instead of random.

w = np.zeros(dim).reshape(dim, 1)

b = 0.0

Learn the parameters for the model by minimizing the cost

Forward propagation of a training pattern's input through the neural network in order to generate the propagation's output activations.

Forward propagation

Compute activation:

A = sigmoid(np.dot(w.T, X) + b)

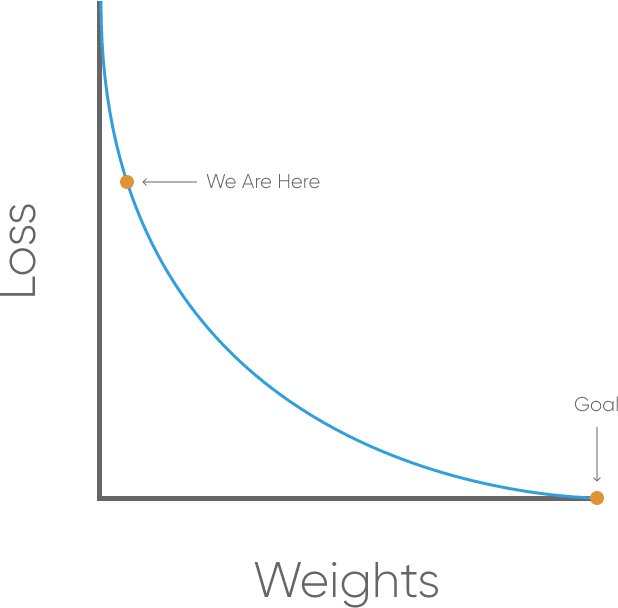
Compute cost:

cost = - 1/m \* np.sum(Y \* np.log(A) + (1 - Y) \* np.log(1 - A))

Backward propagation

Backward propagation of the propagation's output activations through the neural network using the training pattern target in order to generate the deltas of all output and hidden neurons.

The simplest form of calculating the gradients can be shown in the following 2D graph

[](https://res.cloudinary.com/practicaldev/image/fetch/s--ltFBiNlb--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/blog.kabir.ml/img/machine-learning/weightToLoss.svg)

Here we notice that the slope of where we are is negative so we must change the weight by – the derivative of the function times a learning rate.

Our goal is to get it as close as we can to 0. That means we will need to have close to no loss at all. As we are training our network, all we are doing is minimizing the loss.To figure out which direction to alter our weights, we need to find the rate of change of our loss with respect to our weights. In other words, we need to use the derivative of the loss function to understand how the weights affect the input.

This method is known as **gradient descent**. By knowing which way to alter our weights, our outputs can only get more accurate.

A few derivatives of the most popular used activation functions are the following:

Derivation of Sigmoid activation function:

* g(z) = 1 / (1 + np.exp(-z))
* g'(z) = (1 / (1 + np.exp(-z))) \* (1 - (1 / (1 + np.exp(-z))))
* g'(z) = g(z) \* (1 - g(z))

Derivation of Tanh activation function:

* g(z) = (e^z - e^-z) / (e^z + e^-z)
* g'(z) = 1 - np.tanh(z)^2 = 1 - g(z)^2

Derivation of RELU activation function:

* g(z) = np.maximum(0,z)
* g'(z) = { 0 if z < 0
* 1 if z >= 0 }

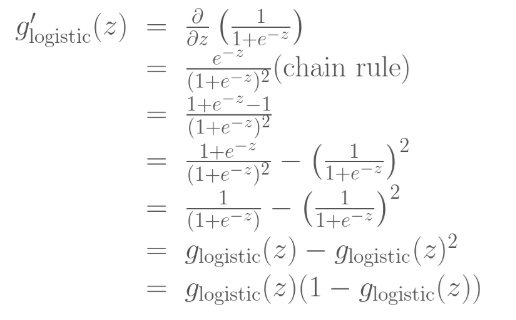
Derivation of leaky RELU activation function:

* g(z) = np.maximum(0.01 \* z, z)
* g'(z) = { 0.01 if z < 0
* 1 if z >= 0 }

In out logistic regression the sigmoid fanction is used:

\Large{\begin{array}{rcl} g_{\text{logistic}}(z) = \frac{1}{1 + e^{-z}}\end{array}}

The logistic sigmoid is motivated somewhat by biological neurons and can be interpreted as the probability of an artificial neuron “firing” given its inputs. (It turns out that the logistic sigmoid can also be derived as the maximum likelihood solution to for [logistic regression](http://en.wikipedia.org/wiki/Logistic_regression) in statistics). Calculating the derivative of the logistic sigmoid function makes use of the quotient rule and a clever trick that both adds and subtracts a one from the numerator:



Here we see that g'_{logistic}(z) evaluated at z is simply g_{logistic}(z) weighted by 1-minus-g_{logistic}(z). This turns out to be a convenient form for efficiently calculating gradients used in neural networks: if one keeps in memory the feed-forward activations of the logistic function for a given layer, the gradients for that layer can be evaluated using simple multiplication and subtraction rather than performing any re-evaluating the sigmoid function, which requires extra exponentiation.

When we take the derivative of the loss function with respect to W and b we get the following.

Compute grads :

dw = 1/m \* np.dot(X, (A - Y).T)

db = 1/m \* np.sum(A - Y)

The gradient decent algorithm repeats: w = w - alpha \* dw where alpha is the learning rate and dw is the derivative of the loss function with respect to w (the derivative is also the slope of w).

def optimize(w, b, X, Y, num\_iterations, learning\_rate, print\_cost = False)

This function optimizes w and b by running a gradient descent algorithm

Arguments:

w -- weights, a numpy array of size (num\_px \* num\_px \* 3, 1)

b -- bias, a scalar

X -- data of shape (num\_px \* num\_px \* 3, number of examples)

Y -- true "label" vector (containing 0 if non-cat, 1 if cat), of shape (1, number of examples)

num\_iterations -- number of iterations of the optimization loop

learning\_rate -- learning rate of the gradient descent update rule

print\_cost -- True to print the loss every 100 steps

Returns:

params -- dictionary containing the weights w and bias b

grads -- dictionary containing the gradients of the weights and bias with respect to the cost function

costs -- list of all the costs computed during the optimization, this will be used to plot the learning curve.

Here are the derivatives of other activation functions we will use later on when we test other NN algos.

Use the learned parameters to make predictions (on the test set)

def predict(w, b, X, threshold=0.5)

Predict whether the label is 0 or 1 using learned logistic regression parameters (w, b)

Arguments:

w -- weights, a numpy array of size (num\_px \* num\_px \* 3, 1)

b -- bias, a scalar

X -- data of size (num\_px \* num\_px \* 3, number of examples)

Returns:

Y\_prediction -- a numpy array (vector) containing all predictions (0/1) for the examples in X

def model(X\_train, Y\_train, X\_test, Y\_test, num\_iterations = 2000, learning\_rate = 0.5, print\_cost = False): Builds the logistic regression model by calling the helper functions

Arguments:

X\_train -- training set represented by a numpy array of shape (num\_px \* num\_px \* 3, m\_train)

Y\_train -- training labels represented by a numpy array (vector) of shape (1, m\_train)

X\_test -- test set represented by a numpy array of shape (num\_px \* num\_px \* 3, m\_test)

Y\_test -- test labels represented by a numpy array (vector) of shape (1, m\_test)

num\_iterations -- hyperparameter representing the number of iterations to optimize the parameters

learning\_rate -- hyperparameter representing the learning rate used in the update rule of optimize()

print\_cost -- Set to true to print the cost every 100 iterations

Returns:

d -- dictionary containing information about the model.

Initialize parameters with zeros

w, b = initialize\_with\_zeros(X\_train.shape[0])

Gradient descent

parameters, grads, costs = optimize(w, b, X\_train, Y\_train, num\_iterations = num\_iterations, learning\_rate = learning\_rate, print\_cost = print\_cost)

Retrieve parameters w and b from dictionary "parameters"

w = parameters["w"]

b = parameters["b"]

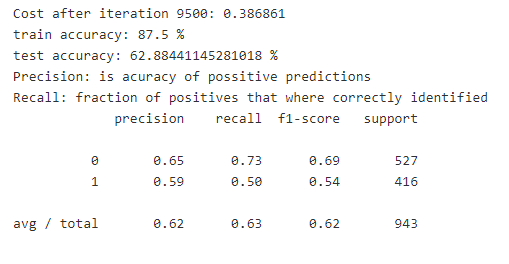
Predict test/train set examples

Y\_prediction\_test = predict(w, b, X\_test)

Y\_prediction\_train = predict(w, b, X\_train)

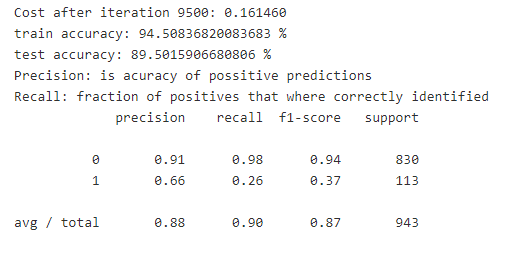
## Analyse results

Kitchen



Precision – Accuracy of positive predictions: we're only getting 62% accuracy, which is better than chance (50%) but still not a good skore. Recall (aka sensitivity or true positive rate): 63%. Fraction of positives that were correctly identified.F1 Score – A helpful metric for comparing two classifiers. F1 Score takes into account precision and the recall.- is 62%. The metric is created by finding the harmonic mean of precision and recall.F1 = 2 x (precision x recall)/(precision + recall)

Bedroom



Precision – Accuracy of positive predictions: we're getting almost 90% accuracy!. This score is the best among the tested rooms. Recall is as high as 90%, very high as well, as the precision outcome.

Bathroom

### 

For bathroom the results in terms of precision and recall are not very high but since they are close to 70% they are high enough to be considered as good scores.

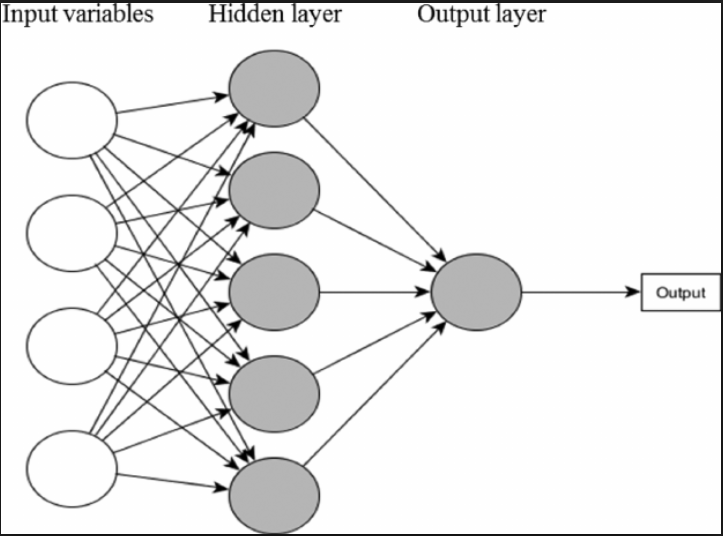
Living Room

### 

### Recall and precision numbers are almost 80% which can be considered as a good score of accurate positive predictions and fraction of positives that were correctly identified.

# Neural Network with hidden layer

We can simulate logistic regression with a neural network that has one hidden layer with a single hidden node and the identity activation function, and a single output node with the logistic sigmoid activation function.



In essence, a neural network is a collection of neurons connected by synapses. This collection is organized into three main layers: the input layer, the hidden layer, and the output layer. We can have many hidden layers, in our analysis below we will add one hidden layer with 5 neurons.

In logistic regression we had:

* X1 \
* X2 ==> z = XW + B ==> a = Sigmoid(z) ==> l(a,Y)
* X3 /

In neural networks with one layer we will have:

* X1 \
* X2 => z1 = XW1 + B1 => a1 = Sigmoid(a1) => z2 = a1W2 + B2 => a2 = Sigmoid(z2) => l(a2,Y)
* X3 /

In addition, so far we are using sigmoid, but in some cases other functions can be a lot better. Sigmoid can lead us to gradient decent problem where the updates are so low.

Sigmoid activation function range is [0,1] A = 1 / (1 + np.exp(-z)) # Where z is the input matrix

Tanh activation function range is [-1,1] (Shifted version of sigmoid function). In NumPy we can implement Tanh using one of these methods:

A = (np.exp(z) - np.exp(-z)) / (np.exp(z) + np.exp(-z)) - Where z is the input matrix

or

A = np.tanh(z) # Where z is the input matrix

It turns out that the tanh activation usually works better than sigmoid activation function for hidden units because the mean of its output is closer to zero, and so it centres the data better for the next layer.

Therefore, the activation function for the hidden layer will be tanh and the activation function for the output will again be a sigmoid.

## Key Steps

initialize\_parameters

W1 -- weight matrix of shape (n\_h, n\_x)

b1 -- bias vector of shape (n\_h, 1)

W2 -- weight matrix of shape (n\_y, n\_h)

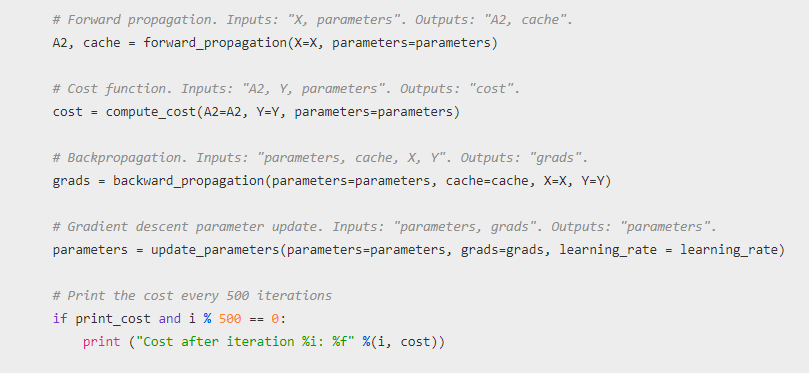
b2 -- bias vector of shape (n\_y, 1)

Implement Forward Propagation to calculate A2 (probabilities)

Compute the cross-entropy cost

Implement the backward propagation having as arguments X -- input data of shape, Y -- "true" labels vector of shape and returns grads.

Updates parameters using the gradient descent update rule.



Computes probabilities using forward propagation, and classifies to 0/1 using 0.5 as the threshold.

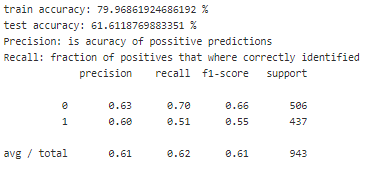


Split training and test set

Run test

Calculate results

The results using one hidden layer are very similar to the ones using logistic regression. Below is the results observed for Kitchen.



So as to improve the results we will use what is industry recemented for I mage processing.

# Convolution Neural networks

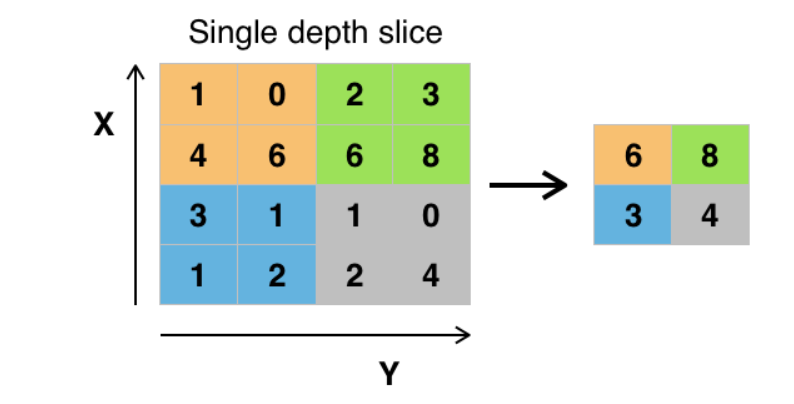
### 

The image shows that we can feed an image as an input to the network, which goes through multiple convolutions, subsampling, a fully connected layer and finally outputs something.

But what are all these concepts?

The convolution layer computes the output of neurons that are connected to local regions or receptive fields in the input, each computing a dot product between their weights and a small receptive field to which they are connected to in the input volume. Each computation leads to extraction of a feature map from the input image. In other words, imagine you have an image represented as a 5x5 matrix of values, and you take a 3x3 matrix and slide that 3x3 window or kernel around the image. At each position of that matrix, you multiply the values of your 3x3 window by the values in the image that are currently being covered by the window. As a result, we 'll get a single number that represents all the values in that window of the images. We use this layer to filtering: as the window moves over the image, you check for patterns in that section of the image. This works because of filters, which are multiplied by the values outputted by the convolution.

The objective of subsampling is to get an input representation by reducing its dimensions, which helps in reducing overfitting. One of the techniques of subsampling is max pooling. With this technique, you select the highest pixel value from a region depending on its size. In other words, max pooling takes the largest value from the window of the image currently covered by the kernel. For example, you can have a max-pooling layer of size 2 x 2 will select the maximum pixel intensity value from 2 x 2 region. You're right to think that the pooling layer then works a lot like the convolution layer! You also take a kernel or a window and move it over the image; The only difference is the function that is applied to the kernel and the image window isn't linear.



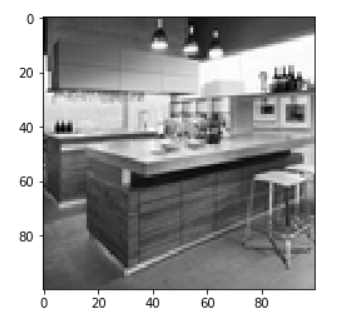
The objective of the fully connected layer is to flatten the high-level features that are learned by convolutional layers and combining all the features. It passes the flattened output to the output layer where you use a softmax classifier or a sigmoid to predict the input class label.

## Importing the Data and Analyzing the data

We will convert each image into greyscale of size 100 x 100 and then into a matrix of size 100 x 100 x 1 which is fed into the network.

img = np.asarray(Image.open(pic\_url).convert('L')).reshape(100, 100, 1)

imshow(Image.open(pic\_url).convert('LA'))



In one-hot encoding, you convert the categorical data into a vector of numbers. The reason why you convert the categorical data in one hot encoding is that machine learning algorithms cannot work with categorical data directly. You generate one boolean column for each category or class. Only one of these columns could take on the value 1 for each sample. Hence, the term one-hot encoding.

For your problem statement, the one hot encoding will be a row vector, and for each image, it will have a dimension of 1 x 10. The important thing to note here is that the vector consists of all zeros except for the class that it represents, and for that, it is 1.

classes = np.unique(Y\_)

nClasses = len(classes)

Total number of outputs : 5

Output classes: [0 1 2 3]

Change the labels from categorical to one-hot encoding

train\_Y\_one\_hot = to\_categorical(train\_Y)

test\_Y\_one\_hot = to\_categorical(test\_Y)

Original label: 2

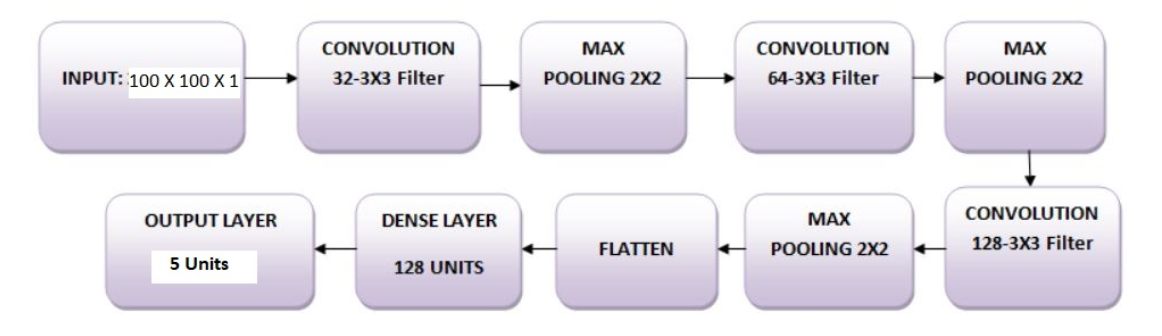
After conversion to one-hot: [0. 0. 1. 0.]

## Architecture of the Model

We will use three convolutional layers:

* The first layer will have 32-3 x 3 filters,
* The second layer will have 64-3 x 3 filters and
* The third layer will have 128-3 x 3 filters.

In addition, there are three max-pooling layers each of size 2 x 2.



## Model the Data

### You will use a batch size of 64 will train the network for 10 epochs.

### batch\_size = 64

### epochs = 10

### num\_classes = nClasses

### We will be using the Leaky ReLU as the activation function

Sigmoid or Tanh function disadvantage is that if the input is too small or too high, the slope will be near zero which will cause us the gradient decent problem.

One of the popular activation functions that solved the slow gradient decent is the RELU function. RELU = max(0,z) # so if z is negative the slope is 0 and if z is positive the slope remains linear.

### 

### Next, you'll add the max-pooling layer with MaxPooling2D() and so on. The last layer is a Dense layer that has a softmax activation function with 10 units, which is needed for this multi-class classification problem.

### 

## Compile the Model

### 

## Train the Model

### 

## Model Evaluation

### 

Test loss: 1.1154791611335773

Test accuracy: 0.692307692724508

The model is overfitting, as the validation loss is 1.2 and the validation accuracy is a 69%.  
Overfitting gives an intuition that the network has memorized the training data very well but is not guaranteed to work on unseen data, and that is why there is a difference in the training and validation accuracy.

### 

From the above two plots, we can see that the validation accuracy almost became stagnant after 4-5 epochs and rarely increased at certain epochs.

The validation loss shows that this is the sign of overfitting, similar to validation accuracy it linearly decreased but after 4-5 epochs, it started to increase. This means that the model tried to memorize the data and succeeded.

With this in mind, it's time to introduce some dropout into our model and see if it helps in reducing overfitting.

## Adding Dropout into the Network

## We can add a dropout layer to overcome the problem of overfitting to some extent. Dropout randomly turns off a fraction of neurons during the training process, reducing the dependency on the training set by some amount. How many fractions of neurons you want to turn off is decided by a hyperparameter, which can be tuned accordingly. This way, turning off some neurons will not allow the network to memorize the training data since not all the neurons will be active at the same time and the inactive neurons will not be able to learn anything.

### 

Test loss: 0.8087572636026324

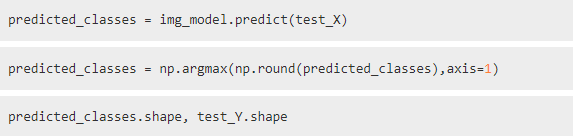
Test accuracy: 0.6899766905324443

### Looks like adding Dropout in our model worked, even though the test accuracy did not improve significantly but the test loss decreased compared to the previous results.

## Predict Labels

Since the predictions you get are floating point values, it will not be feasible to compare the predicted labels with true test labels. So, you will round off the output which will convert the float values into an integer. Further, you will use np.argmax() to select the index number which has a higher value in a row.

For example, let's assume a prediction for one test image to be 0 1 0 0 0 , the output for this should be a class label 1.



bathroom: 0

bedroom: 1

kitchen: 2

livingroom: 3

Results:

65.967366 prcnt correct labels

## Examples of correctly predicted cases

### 

## Examples of incorrectly predicted cases

### 

## Classification Report

### target\_names = ["Class {}".format(i) for i in range(num\_classes)]

### print(classification\_report(test\_Y, predicted\_classes, target\_names=target\_names))

### 

## Adding Dropout into the Network, more apochs, smaller alpha

batch\_size = 50

epochs = 33

num\_classes = nClasses

Compile:

### 

### Train model:

### 

### Evaluate

### 

Test loss: 0.8697960506786

Test accuracy: 0.7296037298816067

Of the 100 cases that have been tested, the test could identify correctly 73 cases.