**Understanding Convolutional Neural Networks (CNNs) in Depth**

[[Koushik](https://medium.com/@koushikkushal95?source=post_page-----d18e299bb438--------------------------------)](https://medium.com/@koushikkushal95?source=post_page-----d18e299bb438--------------------------------)

[Koushik](https://medium.com/@koushikkushal95?source=post_page-----d18e299bb438--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F1670d68f972f&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40koushikkushal95%2Funderstanding-convolutional-neural-networks-cnns-in-depth-d18e299bb438&user=Koushik&userId=1670d68f972f&source=post_page-1670d68f972f----d18e299bb438---------------------post_header-----------)

12 min read

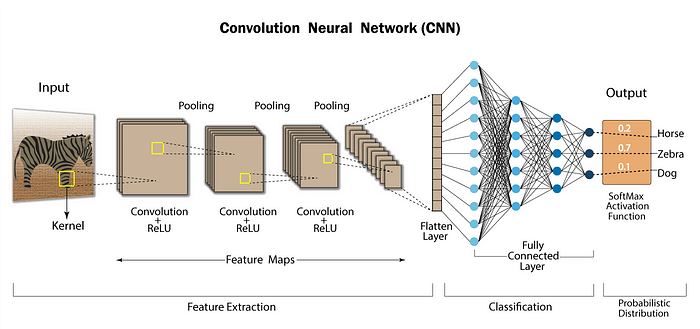
·

Nov 28

181

1

Convolutional Neural Networks skillfully capturing and extracting patterns from data, revealing the hidden artistry within pixels.



A simple classification architecture of Conv-Net

**Introduction:**

Convolutional Neural Networks (CNNs) have revolutionized the field of computer vision, becoming the cornerstone of image and video analysis applications. In this article, we will delve into the key components and operations that make CNNs powerful, exploring concepts like convolution, max-pooling, stride length, padding, upsampling, downsampling and more. Additionally, we’ll discuss a simple CNN model on a dataset using Python and a popular deep learning framework.

Convolutional Neural Networks (CNNs) consist of various types of layers that work together to learn hierarchical representations from input data. Each layer plays a unique role in the overall architecture. Let’s explore the key types of layers found in a typical CNN:

1. **Input Layer:** The input layer is the initial data entry point for the network. In image-based tasks, the input layer represents the pixel values of the image. In the following example, let’s assume we’re working with grayscale images of size 28x28 pixels.

from tensorflow.keras.layers import Input  
  
input\_layer = Input(shape=(28, 28, 1))

2. **Convolutional Layer:** Convolutional layers are the core building blocks of CNNs. These layers apply convolution operations to the input data using learnable filters. These filters scan the input, extracting features such as edges, textures, and patterns

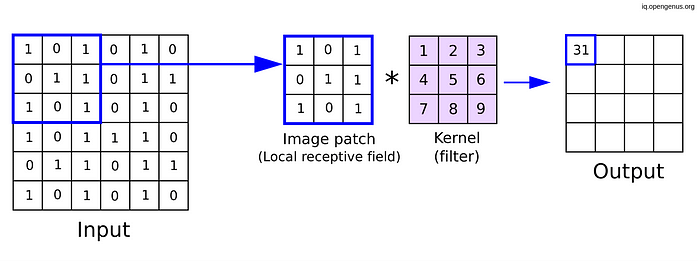
from tensorflow.keras.layers import Conv2D  
  
conv\_layer = Conv2D(filters=32, kernel\_size=(3, 3),  
 activation='relu')(input\_layer)

In the context of Convolutional Neural Networks (CNNs), the terms “kernel” and “filter” are often used interchangeably and refer to the same concept. Let’s break down what these terms mean:

2.1**Kernel:** A kernel is a small matrix used in the convolution operation. It’s a set of learnable weights that are applied to the input data to produce the output feature map. Kernels are the key elements that allow CNNs to automatically learn spatial hierarchies of features within the input data. In image processing, a kernel might be a small matrix like 3x3 or 5x5.

2.2 **Filter:** A filter, on the other hand, is a set of multiple kernels. In most cases, a convolutional layer uses multiple filters to capture different features in the input data. Each filter is convolved with the input to produce a feature map, and the network learns to extract various patterns by adjusting the weights (parameters) of these filters during training.

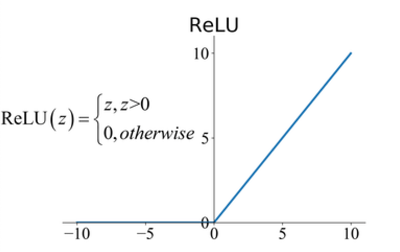
In this example, we’re defining a convolutional layer with 32 filters, each having a 3x3 kernel size. During training, the neural network adjusts the weights (parameters) of these 32 filters to learn different features from the input data. Lets see it wit an image example:



Convolution mechanism.Kernel shape(3X3). imagesource:opengenus

In summary, the kernel is the small matrix that slides or convolves across the input data, and the filter is a set of these kernels used to extract various features from the input, allowing the neural network to learn hierarchical representations.

3. **Activation Layer (ReLU)**: After the convolution operation, an activation function, often Rectified Linear Unit (ReLU), is applied element-wise to introduce non-linearity into the model. ReLU helps the network learn complex relationships and makes the model more expressive. It completely depends upon your use case which activation you will use, in most cases researchers use ReLU, there some activations which can also be used, for example: Leaky ReLU, ELU.

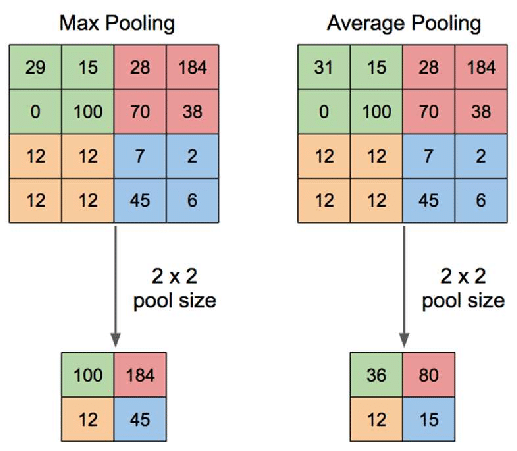


ReLU activation. source: [researchgate](https://www.researchgate.net/figure/Commonly-used-activation-functions-a-Sigmoid-b-Tanh-c-ReLU-and-d-LReLU_fig3_335845675" \t "_blank)

Implementing the Rectified Linear Unit (ReLU) function in Python is quite straightforward. ReLU is an activation function commonly used in neural networks to **introduce non-linearity**. Here’s a simple Python implementation:

def relu(x):  
 return max(0, x)

4. **Pooling Layer**: Pooling layers (e.g., **MaxPooling** or **AveragePooling**) reduce the spatial dimensions of the feature maps generated by the convolutional layers. MaxPooling, for instance, selects the maximum value from a group of values, focusing on the most salient features.



Max Pooling — Average Pooling. source: [researchgate](https://www.researchgate.net/figure/Illustration-of-Max-Pooling-and-Average-Pooling-Figure-2-above-shows-an-example-of-max_fig2_333593451" \t "_blank)

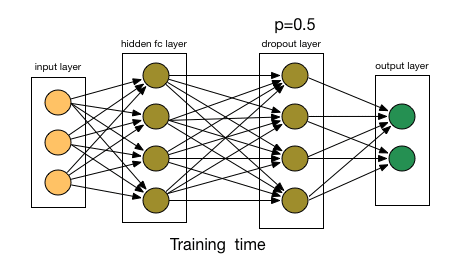
Pooling layers reduce spatial dimensions. MaxPooling is commonly used:

from tensorflow.keras.layers import MaxPooling2D  
  
pooling\_layer = MaxPooling2D(pool\_size=(2, 2))(conv\_layer)

5. **Fully Connected (Dense) Layer**: Fully Connected layers connect every neuron in **one layer to every neuron** in the **next layer**. These layers are typically found towards the end of the network, transforming the learned features into predictions or class probabilities.Fully connected layers are **typically used towards the end of the network**. For classification tasks:

from tensorflow.keras.layers import Dense, Flatten  
  
flatten\_layer = Flatten()(pooling\_layer)  
dense\_layer = Dense(units=128, activation='relu')(flatten\_layer)

6.**Dropout Layer**: Dropout layers are used for **regularization** to [**prevent overfitting**](https://medium.com/@koushikkushal95/what-is-overfitting-and-underfitting-and-how-to-deal-with-it-step-by-step-d6c335ed086b). During training, random neurons are “dropped out,” meaning they are ignored, forcing the network to learn more robust and generalized features. It help prevent overfitting by randomly ignoring a fraction of input units during training:



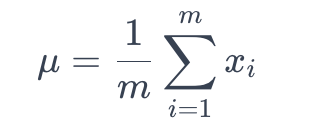
Dropout mechanism. source [nagadakos](https://nagadakos.github.io/2018/09/23/dropout-effect-discussion/" \t "_blank)

from tensorflow.keras.layers import Dropout  
  
dropout\_layer = Dropout(rate=0.5)(dense\_layer)

7. **Batch Normalization Layer**: Batch Normalization (BN) is a technique used in neural networks to stabilize and accelerate the training process. It normalizes the inputs of a layer by adjusting and scaling them during training. The mathematical details behind Batch Normalization involve normalization, scaling, and shifting operations. Let’s delve into the mathematics of Batch Normalization.

Suppose we have a mini-batch of size *m* with *n* features. The input to the Batch Normalization can be summarized as follows:

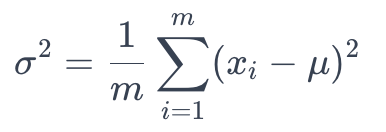
7.1. **Mean Calculation**: Calculate the mean *μ* of the mini-batch for each feature:



Mean of array X

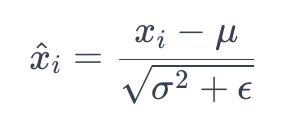
Here, *xi*​ represents the values of the *i*-th feature across the mini-batch.

7.2. **Variance Calculation**: Calculate the variance *σ²* of the mini-batch for each feature:



Variance calculation

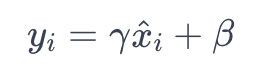
7.3. **Normalize**: Normalize the input by subtracting the mean and dividing by the standard deviation (*σ*)



Normalize in range

Here, *ϵ* is a small constant added to avoid division by zero.

7.4. **Scale and Shift**: Introduce learnable parameters (*γ* and *β*) to scale and shift the normalized values:



Scaling of each batch

Here, *γ* is the scale parameter, and *β* is the shift parameter.

The Batch Normalization operation is typically inserted before the activation function in a neural network layer. It has been shown to have regularization effects and can mitigate issues like internal covariate shift, making training more stable and faster. Here is a simple code, for batch normalization in CNN or any deep neural network.

from tensorflow.keras.layers import BatchNormalization  
  
batch\_norm\_layer = BatchNormalization()(dropout\_layer)

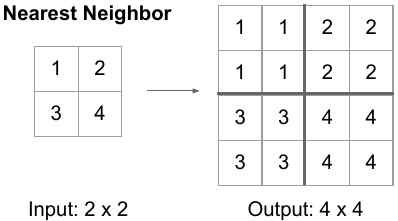
In summary, Batch Normalization normalizes the input, scales and shifts the normalized values, and introduces learnable parameters to allow the network to adapt during training. The use of Batch Normalization has become a standard practice in deep learning architectures.

8. **Flatten Layer**: Flatten layers convert multi-dimensional feature maps into a one-dimensional vector, preparing the data for input into fully connected layers.

flatten\_layer = Flatten()(batch\_norm\_layer)

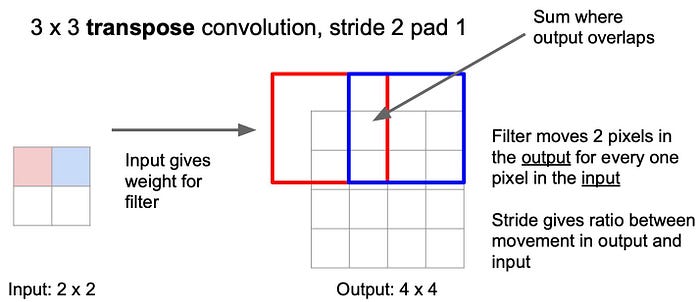
9. **Upsampling Layer**: Upsampling is a technique used in deep learning to increase the spatial resolution of feature maps. It is often employed in tasks like image segmentation and generation. Here are brief descriptions of common types of upsampling methods:

9.1. **Nearest Neighbors (NN) Upsampling**: Nearest Neighbors (NN) upsampling, also known as upsampling by duplication or replication, is a simple and intuitive method. In this approach, each pixel in the input is duplicated or replicated to generate a larger output. While straightforward, NN upsampling may lead to blocky artifacts and a loss of fine details since it does not interpolate between neighboring pixels.



Nearest Neighbor Upsampling.

9.2. **Transposed Convolution (Deconvolution) Upsampling**: Transposed Convolution, often referred to as deconvolution, is a learnable upsampling method. It involves using a convolutional operation with learnable parameters to increase the spatial dimensions of the input. The weights in the transposed convolutional layer are trained during the optimization process, allowing the network to learn upsampling patterns specific to the task.



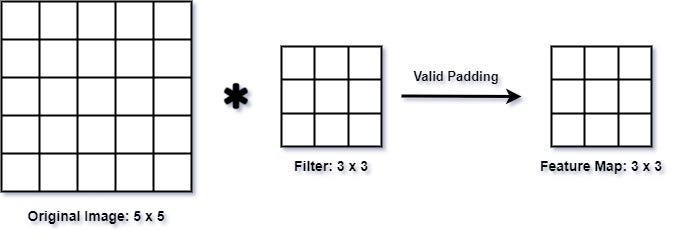
import tensorflow as tf  
from tensorflow.keras.layers import Conv2DTranspose  
  
# Transposed Convolution Upsampling  
transposed\_conv\_upsampling = Conv2DTranspose(filters=32, kernel\_size=(3, 3), strides=(2, 2), padding='same')

Each upsampling method has its advantages and trade-offs, and the choice depends on the specific requirements of the task and the characteristics of the data.

**Padding and stride**

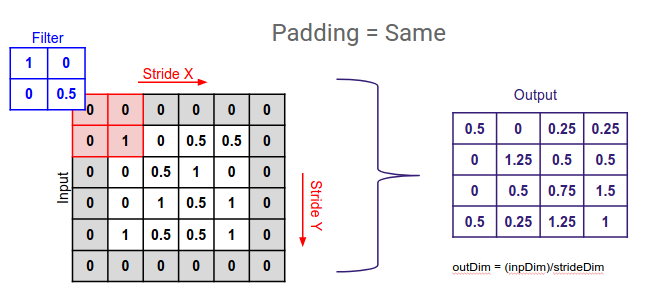
These are crucial concepts in convolutional neural networks (CNNs) that influence the size of the output feature maps after convolution operations. Let’s discuss three types of padding and also explain the concept of stride.

**Valid Padding (No Padding)**: In valid padding, also known as no padding, there is no additional padding added to the input before applying the convolution operation. As a result, the convolution operation is only performed where the filter and the input fully overlap. This often leads to a reduction in the spatial dimensions of the output feature map.



from tensorflow.keras.layers import Conv2D  
  
# Valid Padding  
valid\_padding\_conv = Conv2D(filters=32, kernel\_size=(3, 3),  
 strides=(1, 1), padding='valid')

**Same Padding**: Same padding ensures that the output feature map has the same spatial dimensions as the input. It achieves this by adding zero-padding to the input such that the filter can slide over the input without going outside its boundaries. The amount of padding is calculated to keep the dimensions the same.



from tensorflow.keras.layers import Conv2D  
  
#Padding in Keras  
same\_padding\_conv = Conv2D(filters=32, kernel\_size=(3, 3),   
 strides=(1, 1), padding='same')

**Stride**: Stride defines the step size at which the filter moves across the input during convolution. A larger stride results in a reduction of the spatial dimensions of the output feature map. Stride can be adjusted to control the level of downsampling in the network.

from tensorflow.keras.layers import Conv2D  
  
# Example of Convolution with Stride in Keras  
conv\_with\_stride = Conv2D(filters=32, kernel\_size=(3, 3),   
 strides=(2, 2), padding='same')

In this example, the stride is set to (2, 2), indicating that the filter moves two pixels at a time in both the horizontal and vertical directions. Stride is a critical parameter for controlling the spatial resolution of the feature maps and influencing the receptive field of the network.

In this article, I would like to explore building a simple, Convolutional Neural Network from scratch. Let’s do the most popular classification task of early computer vision task for learning : **Cats vs. Dogs classification.**

This task comprises with a few steps to follow:

**Import Libraries:**

import tensorflow\_datasets as tfds  
import tensorflow as tf  
from tensorflow.keras import layers  
  
import keras  
from keras.models import Sequential,Model  
from keras.layers import Dense,Conv2D,Flatten,MaxPooling2D,GlobalAveragePooling2D  
from keras.utils import plot\_model  
  
import numpy as np  
import matplotlib.pyplot as plt  
import scipy as sp  
import cv2

**Load data: the Cats vs Dogs dataset**

!curl -O https://download.microsoft.com/download/3/E/1/3E1C3F21-ECDB-4869-8368-6DEBA77B919F/kagglecatsanddogs\_5340.zip  
!unzip -q kagglecatsanddogs\_5340.zip  
!ls

The cell below will preprocess the images and create batches before feeding it to our model.

def augment\_images(image, label):  
   
 # cast to float  
 image = tf.cast(image, tf.float32)  
 # normalize the pixel values  
 image = (image/255)  
 # resize to 300 x 300  
 image = tf.image.resize(image,(300,300))  
  
 return image, label  
  
# use the utility function above to preprocess the images  
augmented\_training\_data = train\_data.map(augment\_images)  
  
# shuffle and create batches before training  
train\_batches = augmented\_training\_data.shuffle(1024).batch(32)

**Filter out corrupted images**

When working with lots of real-world image data, corrupted images are a common occurence. Let’s filter out badly-encoded images that do not feature the string “JFIF” in their header.

import os  
  
num\_skipped = 0  
for folder\_name in ("Cat", "Dog"):  
 folder\_path = os.path.join("PetImages", folder\_name)  
 for fname in os.listdir(folder\_path):  
 fpath = os.path.join(folder\_path, fname)  
 try:  
 fobj = open(fpath, "rb")  
 is\_jfif = tf.compat.as\_bytes("JFIF") in fobj.peek(10)  
 finally:  
 fobj.close()  
  
 if not is\_jfif:  
 num\_skipped += 1  
 # Delete corrupted image  
 os.remove(fpath)  
  
print("Deleted %d images" % num\_skipped)

**Generate a**Dataset

image\_size = (300, 300)  
batch\_size = 128  
  
train\_ds, val\_ds = tf.keras.utils.image\_dataset\_from\_directory(  
 "PetImages",  
 validation\_split=0.2,  
 subset="both",  
 seed=1337,  
 image\_size=image\_size,  
 batch\_size=batch\_size,  
)

**Visualize the data**

Here are the first 9 images in the training dataset. As you can see, label 1 is “dog” and label 0 is “cat”.

import matplotlib.pyplot as plt  
  
plt.figure(figsize=(6, 6))  
for images, labels in train\_ds.take(1):  
 for i in range(9):  
 ax = plt.subplot(3, 3, i + 1)  
 plt.imshow(images[i].numpy().astype("uint8"))  
 plt.title(int(labels[i]))  
 plt.axis("off")



Annotated Dataset. Dog: 1 , Cat:0

**Using image data augmentation**

When you don’t have a large image dataset, it’s a good practice to artificially introduce sample diversity by applying random yet realistic transformations to the training images, such as random horizontal flipping or small random rotations. This helps expose the model to different aspects of the training data while slowing down overfitting.

data\_augmentation = keras.Sequential(  
 [  
 layers.RandomFlip("horizontal"),  
 layers.RandomRotation(0.1),  
 ]  
)

Let’s visualize what the augmented samples look like, by applying data\_augmentation repeatedly to the first image in the dataset:

plt.figure(figsize=(6, 6))  
for images, \_ in train\_ds.take(1):  
 for i in range(9):  
 augmented\_images = data\_augmentation(images)  
 ax = plt.subplot(3, 3, i + 1)  
 plt.imshow(augmented\_images[0].numpy().astype("uint8"))  
 plt.axis("off")



Image augmentation. (Ex: Flip, Rotation)

**Configure the dataset for performance**

Let’s apply data augmentation to our training dataset, and let’s make sure to use buffered prefetching so we can yield data from disk without having I/O becoming blocking:

# Apply `data\_augmentation` to the training images.  
train\_ds = train\_ds.map(  
 lambda img, label: (data\_augmentation(img), label),  
 num\_parallel\_calls=tf.data.AUTOTUNE,  
)  
# Prefetching samples in GPU memory helps maximize GPU utilization.  
train\_ds = train\_ds.prefetch(tf.data.AUTOTUNE)  
val\_ds = val\_ds.prefetch(tf.data.AUTOTUNE)

**Build the classifier**

This will look familiar to you because it is almost identical to the previous model we built. The key difference is the output is just one unit that is sigmoid activated. This is because we’re only dealing with two classes.

class CustomModel(Sequential):  
 def \_\_init\_\_(self):  
 super(CustomModel, self).\_\_init\_\_()  
  
 self.add(Conv2D(16, input\_shape=(300, 300, 3), kernel\_size=(3, 3), activation='relu', padding='same'))  
 self.add(MaxPooling2D(pool\_size=(2, 2)))  
  
 self.add(Conv2D(32, kernel\_size=(3, 3), activation='relu', padding='same'))  
 self.add(MaxPooling2D(pool\_size=(2, 2)))  
  
 self.add(Conv2D(64, kernel\_size=(3, 3), activation='relu', padding='same'))  
 self.add(MaxPooling2D(pool\_size=(2, 2)))  
  
 self.add(Conv2D(128, kernel\_size=(3, 3), activation='relu', padding='same'))  
 self.add(GlobalAveragePooling2D())  
 self.add(Dense(1, activation='sigmoid'))  
  
# Instantiate the custom model  
model = CustomModel()  
  
# Display the model summarymodel.summary()

The loss can be adjusted from last time to deal with just two classes. For that, we pick binary\_crossentropy.

# Training will take around 30 minutes to complete using a GPU.   
# If you haven't GPU in your local machine, feel free to use   
# Google colabaratory to get GPU access.  
  
model.compile(loss='binary\_crossentropy',  
 metrics=['accuracy'],  
 optimizer=tf.keras.optimizers.RMSprop(lr=0.001))  
model.fit(train\_ds,  
 epochs=25,  
 validation\_data=val\_ds,)

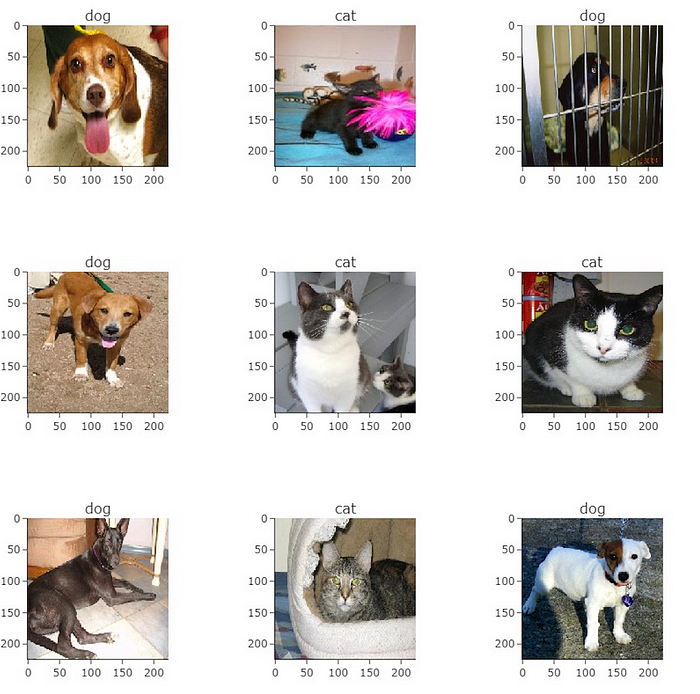
**Testing the Model**

Let’s download a few images and see how the class activation maps look like.

!wget -O cat1.jpg https://storage.googleapis.com/laurencemoroney-blog.appspot.com/MLColabImages/cat1.jpg  
!wget -O cat2.jpg https://storage.googleapis.com/laurencemoroney-blog.appspot.com/MLColabImages/cat2.jpg  
!wget -O catanddog.jpg https://storage.googleapis.com/laurencemoroney-blog.appspot.com/MLColabImages/catanddog.jpg  
!wget -O dog1.jpg https://storage.googleapis.com/laurencemoroney-blog.appspot.com/MLColabImages/dog1.jpg  
!wget -O dog2.jpg https://storage.googleapis.com/laurencemoroney-blog.appspot.com/MLColabImages/dog2.jpg

# utility function to preprocess an image and show the CAM  
def convert\_and\_classify(image):  
  
 # load the image  
 img = cv2.imread(image)  
  
 # preprocess the image before feeding it to the model  
 img = cv2.resize(img, (300,300)) / 255.0  
  
 # add a batch dimension because the model expects it  
 tensor\_image = np.expand\_dims(img, axis=0)  
  
 # get the features and prediction  
 features,results = cam\_model.predict(tensor\_image)  
   
 # generate the CAM  
 show\_cam(tensor\_image, features, results)  
  
convert\_and\_classify('cat1.jpg')  
convert\_and\_classify('cat2.jpg')  
convert\_and\_classify('catanddog.jpg')  
convert\_and\_classify('dog1.jpg')  
convert\_and\_classify('dog2.jpg')

**Output**



Predicted output of classification model.

Thank you!

I wanted to take a moment to express my sincere gratitude for taking the time to read my article. Your engagement and interest mean a lot to me.

Feel free to get the web app based on this classification [here](https://huggingface.co/spaces/Koushikl0l/catsVsdogs).