Convolutional Neural Networks with Tensorflow

Ashu Prasad Feb 14, 2020 · 16 min read ★

Photo by Daniil Kuželev on Unsplash

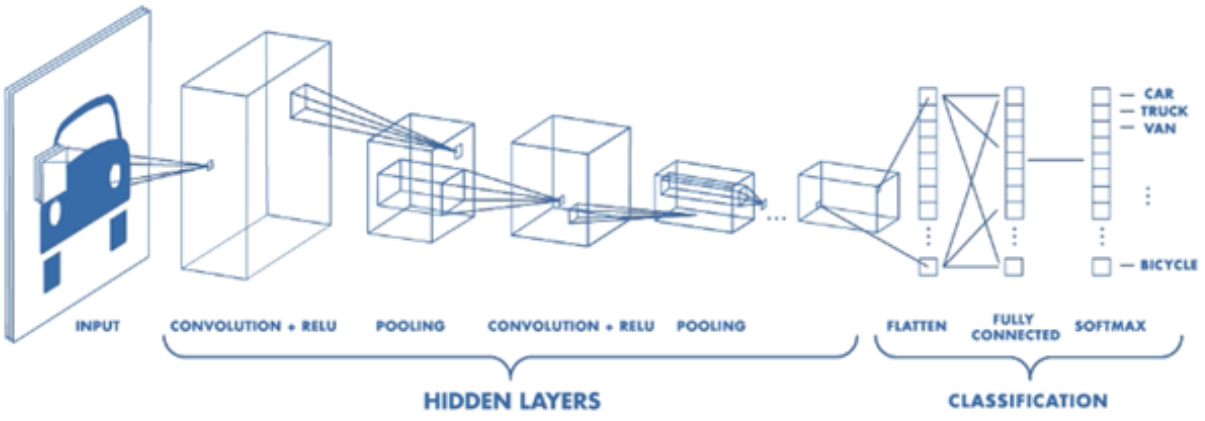
Convolutional Neural Network or ConvNets is a special type of neural network that is used to analyze and process images. It derives it's name from the '**Convolutional**' layer that it employs as a filter. This filters the images fed to it of specific features that is then activated. A feature map is therefore generated by passing the images through these filters to detect particular features, it's position in the image and it's strength that is relevant to the class.

The innovative thing of such neural nets is it's ability to learn to develop these filters automatically, that capture a specific set of features from the input image and classify the image on the basis of the captured features.

These ConvNets have an extensive application in image classification, recommendation systems, image and video recognition, etc.

ConvNets are typically a combination of alternating layers of convolutions with non-linear activation functions and max pooling layers followed by fully connected dense layers.

To give an idea of the architecture of these networks, here's a picture.



Let's understand this by delving right into the course!

We have our dataset in a zipped folder by the name *'cats_and_dogs_filtered.zip'*. To extract it, we'll have to implement the following code.

```
1 import os
2 import zipfile
3
4 local_zip = '/tmp/cats_and_dogs_filtered.zip'
5
6 zip_ref = zipfile.ZipFile(local_zip, 'r')
7
8 zip_ref.extractall('/tmp')
9 zip_ref.close()
```

This will extract the contents in the base directory '*/tmp/cats_and_dogs_filtered*' which contains the training and validation subdirectories which in turn contain cats and dogs subdirectories.

You'll notice that we didn't explicitly label the images as cats and dogs. We'll be going through the **ImageGenerator** later in the article. The **ImageGenerator** can directly read images from subdirectories and label them for you from the name of the subdirectory.

For example, you have a 'train' directory which further contains 'cats' and 'dogs' directory. **ImageGenerator** will label the images present in these subdirectories appropriately.

We define each of these directories.

```
1 base_dir = '/tmp/cats_and_dogs_filtered'
2
3 train_dir = os.path.join(base_dir, 'train')
4 validation_dir = os.path.join(base_dir, 'validation')
5
6 # Directory with our training cat/dog pictures
7 train_cats_dir = os.path.join(train_dir, 'cats')
8 train_dogs_dir = os.path.join(train_dir, 'dogs')
9
10 # Directory with our validation cat/dog pictures
11 validation_cats_dir = os.path.join(validation_dir, 'cats')
12 validation_dogs_dir = os.path.join(validation_dir, 'dogs')
13
```

```
train_cat_fnames = os.listdir( train_cats_dir )
train_dog_fnames = os.listdir( train_dogs_dir )

print(train_cat_fnames[:10])
print(train_dog_fnames[:10])
```

We get an output like this…

```
['cat.829.jpg', 'cat.189.jpg', 'cat.432.jpg', 'cat.647.jpg', 'cat.295.jpg', 'cat.496.jpg'
['dog.160.jpg', 'dog.216.jpg', 'dog.848.jpg', 'dog.99.jpg', 'dog.936.jpg', 'dog.67.jpg',
```

So according to the dataset we have, we've got a total of:

```
print('total training cat images :', len(os.listdir(      train_cats_dir ) ))
print('total training dog images :', len(os.listdir(      train_dogs_dir ) ))

print('total validation cat images :', len(os.listdir( validation_cats_dir ) ))
print('total validation dog images :', len(os.listdir( validation_dogs_dir ) ))
```

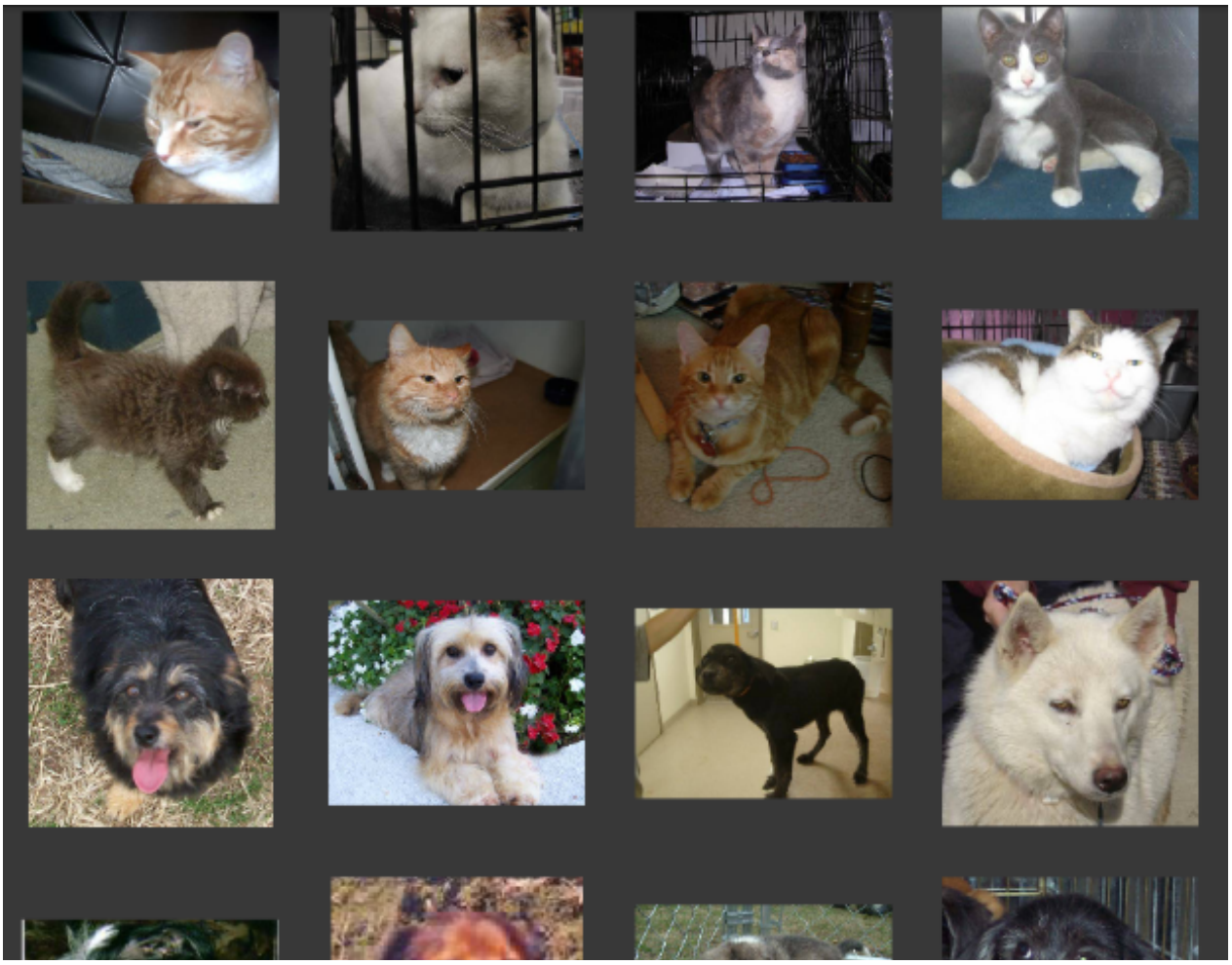
```
total training cat images : 1000
total training dog images : 1000
total validation cat images : 500
total validation dog images : 500
```

We can view our images by implementing the following code:-

First we configure our matplotlib parameters.

```
%matplotlib inline

import matplotlib.image as mpimg
import matplotlib.pyplot as plt

# Parameters for our graph; we'll output images in a 4x4 configuration
nrows = 4
ncols = 4

pic_index = 0 # Index for iterating over images

batch_size = 8 # No. of pictures to load in a batch
```

```
# Set up matplotlib fig, and size it to fit 4x4 pics
fig = plt.gcf()
fig.set_size_inches(ncols*4, nrows*4)

pic_index+=batch_size

next_cat_pix = [os.path.join(train_cats_dir, fname)
                for fname in train_cat_fnames[ pic_index-batch_size:pic_index]
               ]

next_dog_pix = [os.path.join(train_dogs_dir, fname)
                for fname in train_dog_fnames[ pic_index-batch_size:pic_index]
               ]

for i, img_path in enumerate(next_cat_pix+next_dog_pix):
  # Set up subplot; subplot indices start at 1
  sp = plt.subplot(nrows, ncols, i + 1)
  sp.axis('Off') # Don't show axes (or gridlines)

  img = mpimg.imread(img_path)
  plt.imshow(img)

plt.show()
```



Building the model

Images present in the dataset are in a variety of shape and sizes. For a neural network to be trained on these images, we need to have them in a specific shape.

For a greyscale image, we have a colour depth of 1 byte i.e. 8 bits. For the set of images we have in our dataset, there is a colour depth of 3 bytes i.e. 24 bits as they are in RGB.

We add a few of convolutional layers initially in our model and then flatten it before being sent to a densely connected layers.

Since this is a binary classification model, we can add a **'Sigmoid'** function in the last layer.

```
model = tf.keras.models.Sequential([
    # Note the input shape is the desired size of the image 150x150 with 3 bytes color
    tf.keras.layers.Conv2D(16, (3,3), activation='relu', input_shape=(150, 150, 3)),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(32, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2,2),
    # Flatten the results to feed into a DNN
    tf.keras.layers.Flatten(),
    # 512 neuron hidden layer
    tf.keras.layers.Dense(512, activation='relu'),
    # Only 1 output neuron. It will contain a value from 0-1 where 0 for 1 class
    # ('cats') and 1 for the other ('dogs')
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

We'll now configure parameters to train and compile our model. We are going to use **Binary Crossentropy** loss as this is a binary classification problem and we have a **sigmoid** activation function in the last layer that has a range of 0 to 1.

We'll be using **RMSprop optimization algorithm** instead of **Stochastic Gradient Descent (SGD)** because RMSprop automates the learning rate tuning for us.

```
from tensorflow.keras.optimizers import RMSprop

model.compile(optimizer=RMSprop(lr=0.001),
```

Why do we implement Conv2D and MaxPooling2D?

The idea of implementing convolutional layers is to reduce the image so that only the features that stand out and those that determine the output are considered. In other words, the convolutions compress the image.

Filter matrix allows one to extract features from an image and feed it to a neural network to learn the most out of an image, mostly allowing us to distinguish one class of image from another. The ***Pooling layer*** effectively compresses an image to make it more manageable and select the features that stand out.

```
import tensorflow as tf

print(tf.__version__)

mnist = tf.keras.datasets.fashion_mnist

(training_images, training_labels), (test_images, test_labels) = mnist.load_data()

training_images=training_images.reshape(60000, 28, 28, 1)
training_images=training_images / 255.0
test_images = test_images.reshape(10000, 28, 28, 1)
test_images=test_images/255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Conv2D(64, (3,3), activation='relu', input_shape=(28, 28, 1)),
  tf.keras.layers.MaxPooling2D(2, 2),
  tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
  tf.keras.layers.MaxPooling2D(2,2),
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.summary()

model.fit(training_images, training_labels, epochs=5)

test_loss = model.evaluate(test_images, test_labels)
```

```
training_images=training_images.reshape(60000, 28, 28, 1)
```

The above line of code reshapes the training image into a ***28x28 pixel image***, ***60000 represents number of images***, ***1 represents the number of channels***. Since the above images are grayscale images, there is only 1 channel associated with it.

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Conv2D(64, (3,3), activation='relu', input_shape=(28, 28, 1)),
  tf.keras.layers.MaxPooling2D(2, 2),
```

```
22 ])
```

Again like the previous code snippet, the ***Conv2D*** layer's ***input_shape*** is ***28x28 pixels*** and ***1 channel*** is associated with each image as its a gray scale image. ***64*** convolutions are present and each convolution is a matrix of shape ***3x3***.

MaxPooling2D layer has a pooling matrix of shape ***2x2***.

Conv2D has the ***first parameter*** as ***64*** which denoted the number of ***filters***. The **initial filters in a network is responsible for detecting edges and blobs**. Using **too many filters in the initial layer** is **not required** as there is only so much information extractable from the raw input layer and **most of the filters will be redundant**.

Use 16 or 32 in the initial layers.

The summary of the model gives us an output like this.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 64)        640
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 64)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 64)        36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 1600)              0
_________________________________________________________________
dense (Dense)                (None, 128)               204928
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 243,786
Trainable params: 243,786
Non-trainable params: 0
```

The above code snippet gives us the summary of the entire model.

The first ***Conv2D*** layer's output shape is of ***26x26.*** This isn't a glitch, let's understand why:-

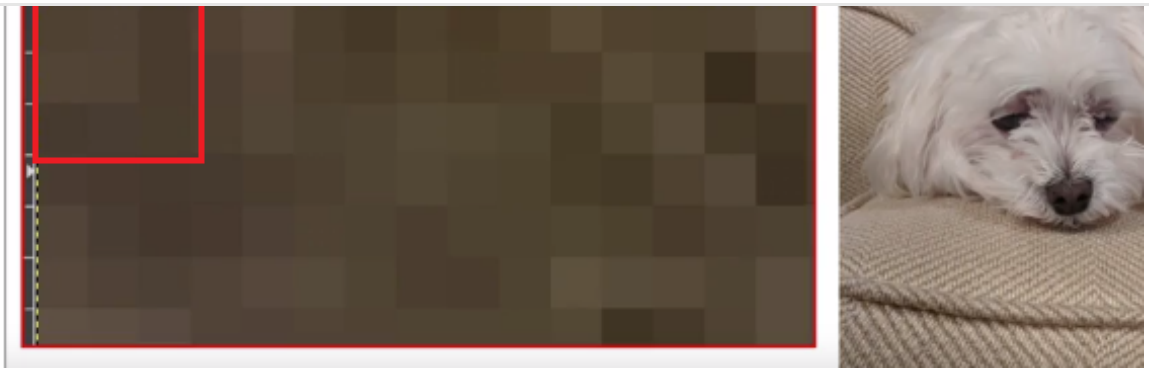


Fig. — 1

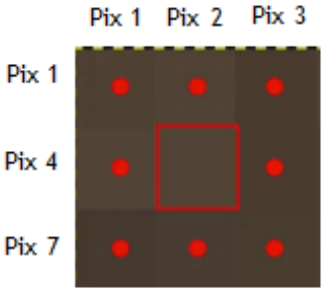
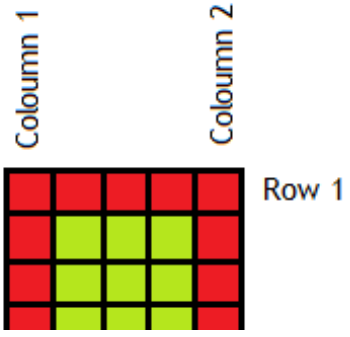


Fig.-2, Let's name each of the nine pixels horizontally as pix1, pix2, pix3...

The ***Conv2D*** layer has a ***filter size of 3x3***. This means it needs to select the pixels having a (pixel above or below) and (pixel to its left or right). The Fig. — 2 exhibits the pixel which is the first likely contender satisfying the criteria in the top-left corner of Fig. — 1.

We can see that there doesn't exist any pixel above pix1, pix2, pix3. Similarly there doesn't exist any pixel to the left of pix1, pix4, pix7. Consequently we need to select pix5 as the first pixel. Similarly in the bottom-left corner, we'll have to select a pixel accordingly. **Therefore we are omitting a pixel row at the top and a pixel row at the bottom; a pixel coloumn to the left and a pixel coloumn to the right.**



2 pixels row wise and 2 pixel coloumn wise have been omitted. Consequently 28x28 pic has been reduced to 26x26.

MaxPooling2D has a ***filter size of 2x2***. It'll **slide over the 26x26 picture** and **select the pixel with max value**. This effectively **reduces the image size by half**. Therefore **26x26 pic becomes a pic size of 13x13**.

The same process continues until we reach the layer ***max_pooling2d_1.*** Here we have a ***convolution window of size 5x5*** and a ***64 convolutions***. Now in the subsequent layer, the convolution is flattened. Therefore we have an output shape of the flatten layer as 1600 **(5x5x64 = 25x64 = 1600)**. The 1600 features now exist as a vector and can be subsequently fed into the dense network in the following layers.

Understanding ImageGenerator

We'll begin this by taking the code snippet from a horse and human classifier.

```
[13]
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# All images will be rescaled by 1./255
train_datagen = ImageDataGenerator(rescale=1/255)

# Flow training images in batches of 128 using train_datagen generator
train_generator = train_datagen.flow_from_directory(
        '/tmp/horse-or-human/',  # This is the source directory for training images
        target_size=(300, 300),  # All images will be resized to 150x150
        batch_size=128,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

```

Here we have a '***rescale'*** parameter. The **value specified by** the argument **'rescale' is the factor by which each pixel value of the image is multiplied**. Eg.- ***A pixel has a value of 223***. 223 gets multiplied by 1/255 and the ***original 223 is replaced by the result 0.87 (223x 1/255 = 0.87)***.

The **data generator** can then be **used with Keras model** to **accept data generators as inputs**: ***fit_generator***, ***evaluate_generator***, ***predict_generator***.

```
[14]
history = model.fit_generator(
      train_generator,
      steps_per_epoch=8,
      epochs=15,
      verbose=1)
```

```python
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# All images will be rescaled by 1./255
train_datagen = ImageDataGenerator(rescale=1/255)
validation_datagen = ImageDataGenerator(rescale=1/255)

# Flow training images in batches of 128 using train_datagen generator
train_generator = train_datagen.flow_from_directory(
        '/tmp/horse-or-human/',  # This is the source directory for training images
        target_size=(300, 300),  # All images will be resized to 150x150
        batch_size=128,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')

# Flow training images in batches of 128 using train_datagen generator
validation_generator = validation_datagen.flow_from_directory(
        '/tmp/validation-horse-or-human/',  # This is the source directory for training images
        target_size=(300, 300),  # All images will be resized to 150x150
        batch_size=32,
        # Since we use binary_crossentropy loss, we need binary labels
        class_mode='binary')
```

```
[17]
history = model.fit_generator(
      train_generator,
      steps_per_epoch=8,  #Should be equal to ceil(number of samples in training
                          #set / batch size)
      epochs=15,
      verbose=1,
      validation_data = validation_generator,
      validation_steps=8) #Should be equal to the number of samples of your
                          #validation dataset divided by the batch size
```

Referencing the above code, we can analyze the training and validation trend of the model.

Let's try to follow the same procedure for our existing 'cats and dogs' dataset.

```python
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# All images will be rescaled by 1./255.
train_datagen = ImageDataGenerator( rescale = 1.0/255. )
test_datagen  = ImageDataGenerator( rescale = 1.0/255. )

# --------------------
# Flow training images in batches of 20 using train_datagen generator
# --------------------
train_generator = train_datagen.flow_from_directory(train_dir,
                                                    batch_size=20,
                                                    class_mode='binary',
                                                    target_size=(150, 150))
# --------------------
# Flow validation images in batches of 20 using test_datagen generator
# --------------------
validation_generator =  test_datagen.flow_from_directory(validation_dir,
                                                         batch_size=20,
```

We train our model by using the **'fit_generator'** method in the training set.

```
history = model.fit_generator(train_generator,
                              validation_data=validation_generator,
                              steps_per_epoch=100,
                              epochs=15,
                              validation_steps=50,
                              verbose=2)
```

The below code allows one to upload files and in google colab and subsequently run it through the model, attempting to predict whether the image is a cat or a dog.

```
import numpy as np

from google.colab import files
from keras.preprocessing import image

uploaded=files.upload()

for fn in uploaded.keys():

  # predicting images
  path='/content/' + fn
  img=image.load_img(path, target_size=(150, 150))

  x=image.img_to_array(img)
  x=np.expand_dims(x, axis=0)
  images = np.vstack([x])

  classes = model.predict(images, batch_size=10)

  print(classes[0])

  if classes[0]>0:
    print(fn + " is a dog")

  else:
    print(fn + " is a cat")
```

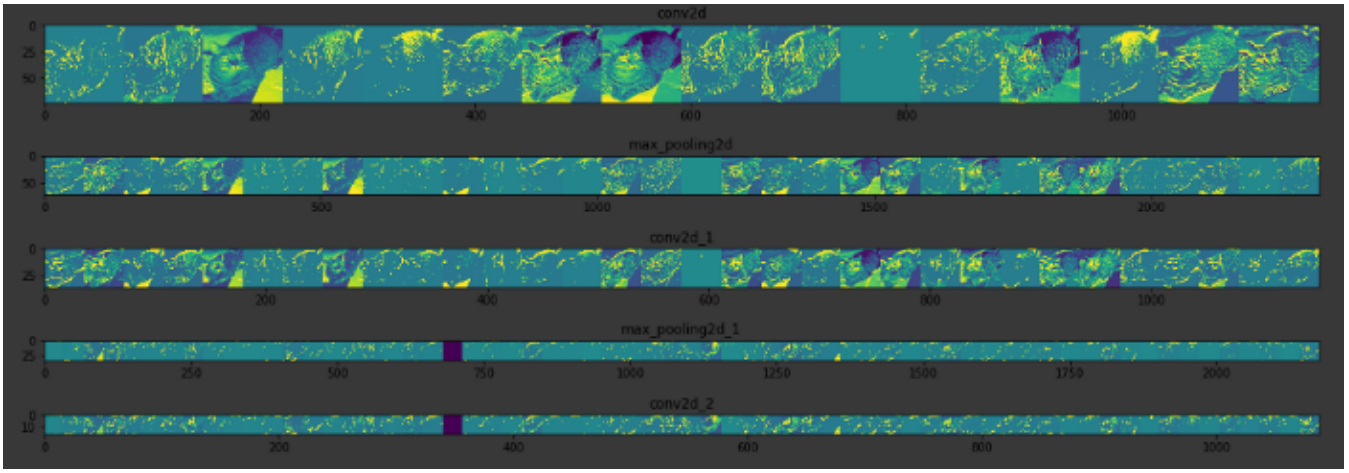
Viewing the output of each layer

Let's try to understand how our convolutional neural net is attempting to predict the class of the images. To do this we need to understand what features our ConvNet is paying attention to in the picture.

```
import numpy as np

import random

from   tensorflow.keras.preprocessing.image import img_to_array,
load_img

# Let's define a new Model that will take an image as input, and
will output

# intermediate representations for all layers in the previous model
after

# the first.

successive_outputs = [layer.output for layer in model.layers[1:]]

#visualization_model = Model(img_input, successive_outputs)

visualization_model = tf.keras.models.Model(inputs = model.input,
outputs = successive_outputs)

# Let's prepare a random input image of a cat or dog from the
training set.

cat_img_files = [os.path.join(train_cats_dir, f) for f in
train_cat_fnames]

dog_img_files = [os.path.join(train_dogs_dir, f) for f in
train_dog_fnames]

img_path = random.choice(cat_img_files + dog_img_files)

img = load_img(img_path, target_size=(150, 150))  # this is a PIL
image

x   = img_to_array(img)                           # Numpy array
with shape (150, 150, 3)

x   = x.reshape((1,) + x.shape)                   # Numpy array
with shape (1, 150, 150, 3)

# Rescale by 1/255

x /= 255.0

# Let's run our image through our network, thus obtaining all
```

```
# These are the names of the layers, so can have them as part of
our plot

layer_names = [layer.name for layer in model.layers]

# ----------------------------------------------------------------
------

# Now let's display our representations

# ----------------------------------------------------------------
------

for layer_name, feature_map in zip(layer_names,
successive_feature_maps):

if len(feature_map.shape) == 4:

#----------------------------------------

# Just do this for the conv / maxpool layers, not the fully-
connected layers

#----------------------------------------

n_features = feature_map.shape[-1]  # number of features in the
feature map

size       = feature_map.shape[ 1]  # feature map shape (1, size,
size, n_features)

# We will tile our images in this matrix

display_grid = np.zeros((size, size * n_features))

#-----------------------------------------------

# Postprocess the feature to be visually palatable

#-----------------------------------------------

for i in range(n_features):

x  = feature_map[0, :, :, i]

x -= x.mean()

x /= x.std ()
```

```
x  = np.clip(x, 0, 255).astype('uint8')

display_grid[:, i * size : (i + 1) * size] = x # Tile each filter
into a horizontal grid

#-----------------

# Display the grid

#-----------------

scale = 20. / n_features

plt.figure( figsize=(scale * n_features, scale) )

plt.title ( layer_name )

plt.grid  ( False )

plt.imshow( display_grid, aspect='auto', cmap='viridis' )
```

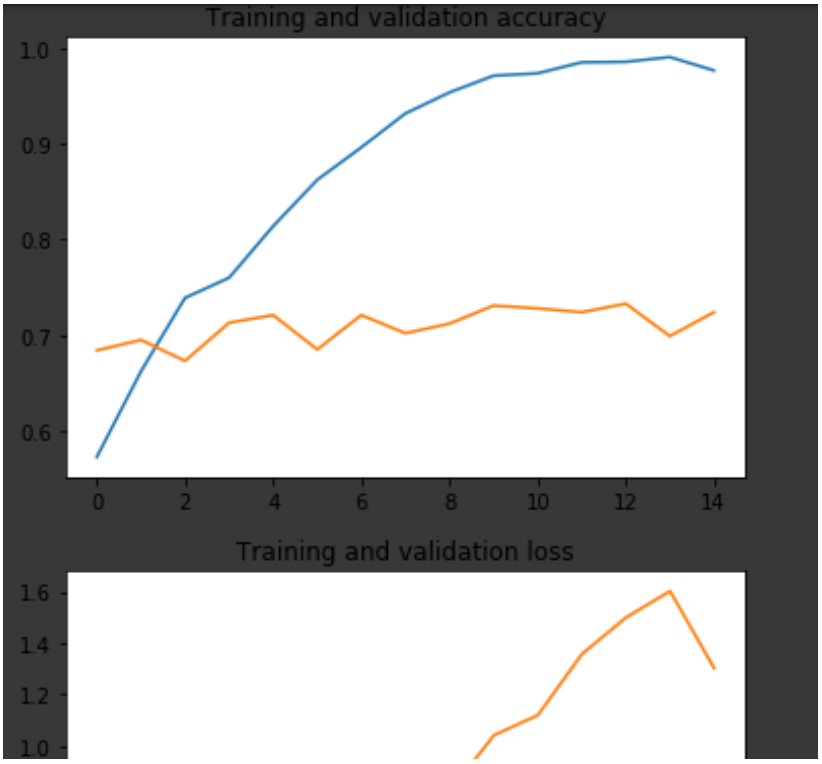


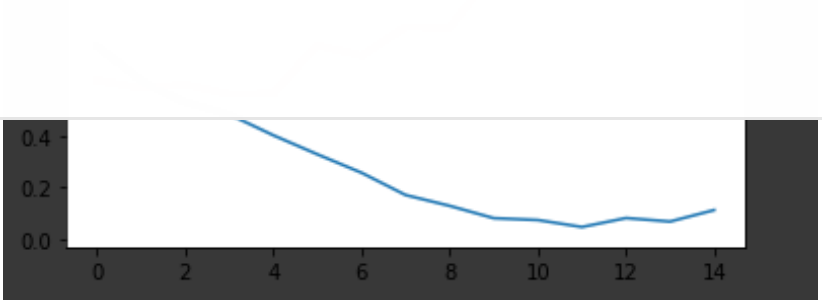
As we can see, we move from a set of pixels where a cat can be clearly seen to an increasingly compact and abstract representation of the same image. As we go downstream we observe that the network starts highlighting what the network is paying attention to. Fewer features from the image can be observed to be highlighted. Most of the features in the image are set to ‘zero’. This is called ‘**sparsity**’ which is a key feature of deep learning.

As we go downstream, the representation carry increasingly less information about the original pixel of the image, but increasingly refined information about the class of

Visualizing the accuracy trend

```
#-----------------------------------------------------------
# Retrieve a list of list results on training and test data
# sets for each training epoch
#-----------------------------------------------------------
acc      = history.history[     'acc' ]
val_acc  = history.history[ 'val_acc' ]
loss     = history.history[    'loss' ]
val_loss = history.history['val_loss' ]

epochs   = range(len(acc)) # Get number of epochs

#------------------------------------------------
# Plot training and validation accuracy per epoch
#------------------------------------------------
plt.plot  ( epochs,     acc )
plt.plot  ( epochs, val_acc )
plt.title ('Training and validation accuracy')
plt.figure()

#------------------------------------------------
# Plot training and validation loss per epoch
#------------------------------------------------
plt.plot  ( epochs,     loss )
plt.plot  ( epochs, val_loss )
plt.title ('Training and validation loss'   )
```





The blue line represents the model's performance on training data and the orange line represents the model's performance on validation data. We observe that our training accuracy goes to about 100% while our validation accuracy is stuck around 70%. Similarly our training loss steadily decreases while our validation loss continues to steadily increase.

This is classic example of **overfitting**. Overfitting happens when a model is exposed to a small number of examples, in this case only 2000. The model when trained on a small training dataset tends to learn irrelevant features that doesn't generalize the data. For example, when a human is given a small set of pictures of lumberjacks and sailors and suppose only lumberjacks in the pictures are wearing caps, a human may falsely associate caps to lumberjacks, i.e. the human may now start thinking that all lumberjacks wear caps as opposed to a sailor.

To overcome this problem, we can simply augment the images i.e. we simply tweak the images to change it a bit.

Image Augmentation

```
train_datagen = ImageDataGenerator(
      rescale=1./255,
      rotation_range=40,
      width_shift_range=0.2,
      height_shift_range=0.2,
      shear_range=0.2,
      zoom_range=0.2,
      horizontal_flip=True,
      fill_mode='nearest')

test_datagen = ImageDataGenerator(rescale=1./255)
```

Now instead of only the **rescale** feature of the ImageDataGenerator, we are implementing image augmentation as well.

Image augmentation helps to deal with overfitting. If images are augmented, it can create new sets of images to train the model on without explicitly getting new images.

However on the downside, if the testing set of images lack in diversity and bears more resemblance to the training set, we may have a case where image augmentation does not reflect us overcoming overfitting problem. Therefore we must have a wide range of images in testing set also.

The options available to us are:

rotation_range — Value in degrees through which we can rotate the picture. Varies between 0 to 180.

width_shift and height_shift — A fraction of total width or height within which to randomly translate pictures vertically or horizontally.

shear_range — For randomly applying shearing transformations.

zoom_range — For randomly zooming inside pictures.

horizontal_flip — For randomly flipping half of the images horizontally. Relevant in real-world pictures. Eg.- A landscape when viewed from one side or after horizontally flipping it seems still to be a natural picture.

fill_mode — Strategy used for filling in newly created pixels, which can appear after rotation or a width height shift.

Transfer learning

Transfer learning is a technique in machine learning that focuses on storing knowledge learnt from solving one type of problem and implementing the solution to some other related type of problem.

We can implement this in Keras using the **layers** API that help us to peek into the layers of the pre-trained model. This helps us to identify which layers we would like to use and which we would like to retrain.

A snapshot of the pre-trained model is used. Parameters can later be loaded into this skeleton model of the pre-trained neural net to turn in into a trained model.

connected layer at the top. By specifying **include_top = False**, we are telling the model to ignore the fully connected layer and get straight to the convolutions. By assigning the **weights** parameter to **None**, we are specifying that we don't want the built-in weights of the model. We can later assign our desired weights by **load_weights** method and load the downloaded weights into the model.

```
from tensorflow.keras.applications.inception_v3 import InceptionV3

local_weights_file = '/tmp/inception_v3_weights_tf_dim_ordering_tf_kernels_notop.h5'

pre_trained_model = InceptionV3(input_shape = (150, 150, 3),
                                include_top = False,
                                weights = None)

pre_trained_model.load_weights(local_weights_file)
```

We now are ready with our pre-trained model. We can iterate through the model layers and lock them, in other words, we specify if the layers are trainable or not.

```
for layer in pre_trained_model.layers:
  layer.trainable = False
```

All the layers in the model have names so we can look into a specific layer. We can take any layer from the pre-trained model and grab it's output.

```
last_layer = pre_trained_model.get_layer('mixed7')

last_output = last_layer.output
```

We'll now take that output and join it with a dense layer since we had ignored the fully connected layer during instantiating our model. We'll then consolidate all the

we'll require.

```
from tensorflow.keras.optimizers import RMSprop

x = layers.Flatten()(last_output)
x = layers.Dense(1024, activation='relu')(x)
x = layers.Dense  (1, activation='sigmoid')(x)

model = Model( pre_trained_model.input, x)
model.compile(optimizer = RMSprop(lr=0.0001),
              loss = 'binary_crossentropy',
              metrics = ['acc'])
```

Once this is done, we'll follow the same procedure as before.

We'll create an Image Data Generator and augment our images.

```
# Add our data-augmentation parameters to ImageDataGenerator
train_datagen = ImageDataGenerator(rescale = 1./255.,
                                   rotation_range = 40,
                                   width_shift_range = 0.2,
                                   height_shift_range = 0.2,
                                   shear_range = 0.2,
                                   zoom_range = 0.2,
                                   horizontal_flip = True)
```

We can then specify the directory containing the training images and make it flow through the generator to apply all the augmentations on the images from the training directory.

```
train_generator = train_datagen.flow_from_directory(
```

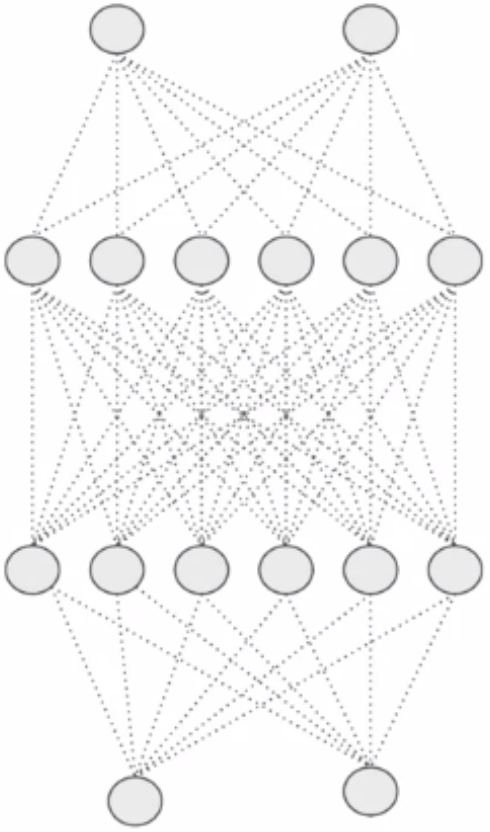
```
                    batch_size = 20,
                    class_mode = 'binary',
                    target_size = (150, 150))
```

The model is then trained as before by passing it through the **fit_generator** method. Here, the model is trained for 100 epochs.

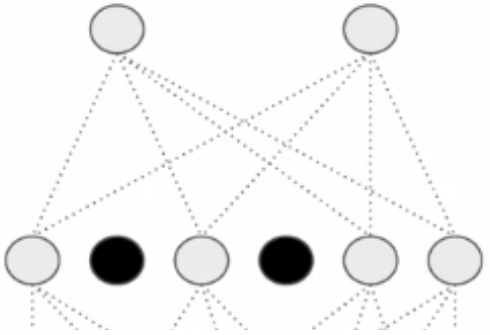
```
history = model.fit_generator(
            train_generator,
            validation_data = validation_generator,
            steps_per_epoch = 100,
            epochs = 100,
            validation_steps = 50,
            verbose = 2)
```

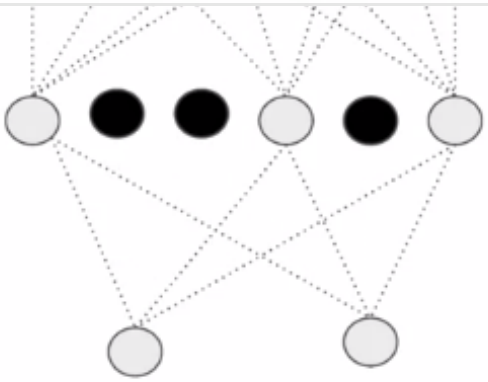
The above is a graph for the accuracy vs training and validation set. The model started off good but subsequently began to diverge from the training. Thus we end up in a different kind of overfitting situation.

To understand this, let's assume that the below diagram is our neural network. The idea behind this kind of error is that **layers in the neural net can sometimes end up having similar weights and can possibly impact each other when our model is attempting to predict the data leading to overfitting**. It's quite often we see such kind of errors in big complex networks.



By implementing **'dropout'**, we are effectively making our model look like this.





The consequence is that the neighbours don't affect each other too much and therefore we can potentially overcome overfitting.

We can implement this in code by adding the **Dropout** layer. The parameter can be anything between 0 and 1. Here we have given our parameter as 0.2 which means we'll be dropping out 20% of our neurons.

```
from tensorflow.keras.optimizers import RMSprop

x = layers.Flatten()(last_output)
x = layers.Dense(1024, activation='relu')(x)
x = layers.Dropout(0.2)(x)
x = layers.Dense  (1, activation='sigmoid')(x)

model = Model( pre_trained_model.input, x)
model.compile(optimizer = RMSprop(lr=0.0001),
              loss = 'binary_crossentropy',
              metrics = ['acc'])
```

When we begin to observe the validation accuracy diverging away from our training accuracy, the problem becomes a good candidate to implement Dropout.

Here's the graph after implementing dropout.

Multiclass classification

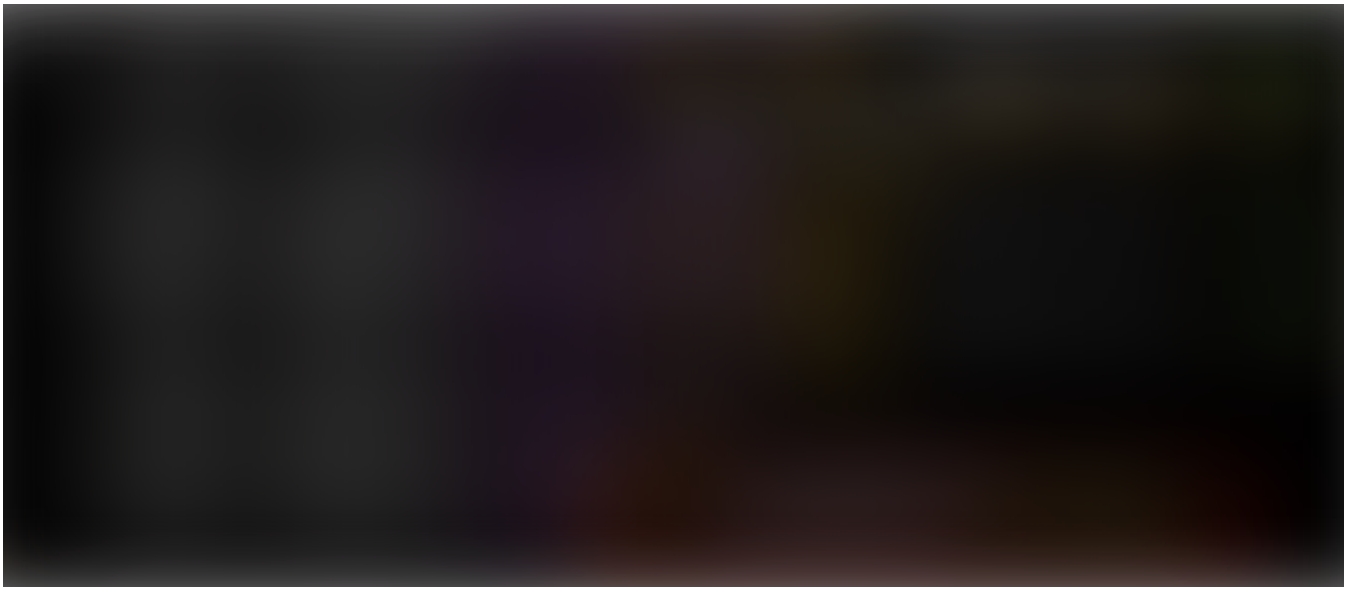
So far we've gone through binary classification problems where we attempted to classify images into two categories. We'll now begin 'multiclass classification' where we attempt to classify images into more than two categories.

The above is the file structure we need to maintain in order to implement what we have learnt for binary classification just as we had maintained the file structure for the cats and dogs classification problem.

Similar to the previous problem, we'll be creating an Image Data Generator and instead of setting the class mode to '**Binary**', we'll be setting it to '**Crossentropy**'.

We'll be changing the model definition next.

For binary classification problem, we'll be having a single neuron with the '**Sigmoid**' activation function for our last layer as this will either be 0 or 1 depending on the prediction of the model.



Here we have 3 neurons in our output layer with the '**Softmax**' activation function.

In the case of **Softmax** function with 3 neurons, one of the neuron will light up when given an input resembling the below image. Note that the summation of all the values add up to 1. We can call these values as the probabilities of each class. The image most likely belongs to the class having the highest probability and therefore that neuron will light up.

There's one more change that we'll be implementing. For the binary classification problem, we had the '**binary_crossentropy**' loss function.

For multi-class classification problem, we'll be having the '**categorical_crossentropy**'

Since in the previous 'cats and dogs' classification problem, we had only two classes to classify data into, we used the '**binary_crossentropy**' loss function. This time we are classifying data into three classes and therefore are using '**categorical_crossentropy**' function. In general, all multi-class classification problems i.e. problems having more than two classes will be using this loss function.

With this we come to the end of the documented course. Over the course of this article, we learnt how to build convolutional neural nets for image classification, implement data augmentation and transfer learning and learnt how to implement multi class classifiers. However we have just scratched the surface of the copious amount of applications that **computer vision** has to offer. Several of these computer vision applications incorporate complex architectures and strategies derived from the basic structure of a convolutional neural net that we have seen in this course.

I hope this motivates you to take up the course and begin experimenting, exploring and implementing these intricacies of convolutional neural nets for yourself.

Thanks for reading this blog. I would appreciate hearing your thoughts on this.