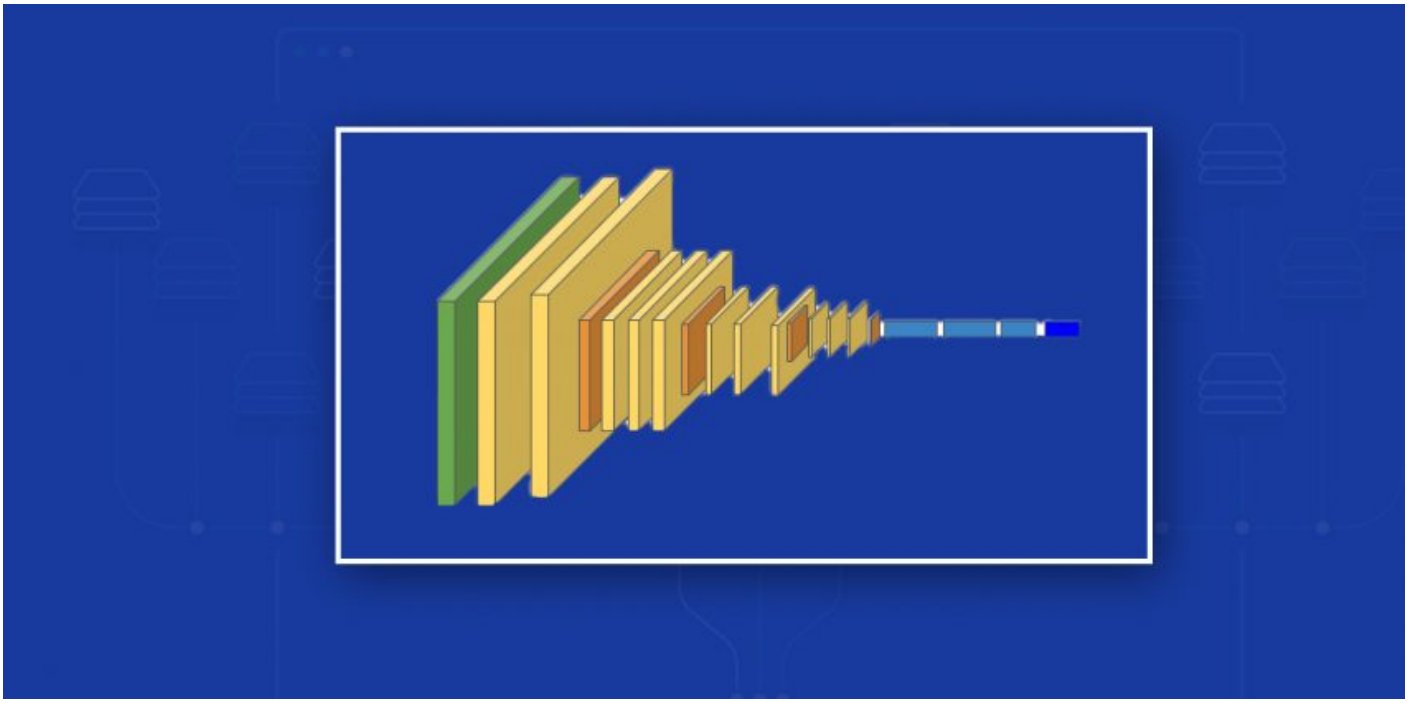
Top 4 Pre-Trained Models for Image Classification with Python Code

COMPUTER VISION IMAGE INTERMEDIATE LISTICLE PYTHON UNSTRUCTURED DATA

Introduction

The human brain can easily recognize and distinguish the objects in an image. For instance, given the image of a cat and dog, within nanoseconds, we distinguish the two, and our brain perceives this difference. In case a machine mimics this behavior, it is as close to Artificial Intelligence as we can get. Subsequently, the field of Computer Vision aims to mimic the human vision system – and there have been numerous milestones that have broken the barriers in this regard.

Moreover, nowadays, machines can easily distinguish between different images, detect objects and faces, and even generate images of people who don't exist! Fascinating, isn't it? One of my first experiences when starting with Computer Vision was the task of Image Classification. This very ability of a machine to distinguish between objects leads to more avenues of research – like distinguishing between people.



The rapid developments in Computer Vision and, by extension – image classification has been further accelerated by the advent of Transfer Learning. To put it simply, Transfer learning allows us to use a pre-existing model, trained on a huge dataset, for our own tasks. Consequently reducing the cost of training new deep learning models and since the datasets have been vetted, we can be assured of the quality.

In Image Classification, there are some very popular datasets that are used across research, industry, and hackathons. The following are some of the prominent ones:

- ImageNet
- CIFAR

- MNIST

and many more.

In this article, I will cover the top 4 pre-trained Image Classification models that are state-of-the-art (SOTA) and are widely used in the industry as well. The individual models can be explained in much more detail, but I have limited the article to give an overview of their architecture and implement it on a dataset.

In case you want to learn computer vision in a structured format, refer to this course- Certified Computer Vision Master's Program

Table of contents

Setting Up the System

Since we started with cats and dogs, let us take up the dataset of Cat and Dog Images. The original training dataset on Kaggle has 25000 images of cats and dogs and the test dataset has 10000 unlabelled images. Since our purpose is only to understand these models, I have taken a much smaller dataset. You can straight-up run this and the rest of the code on Google Colab as well – so let us get started!

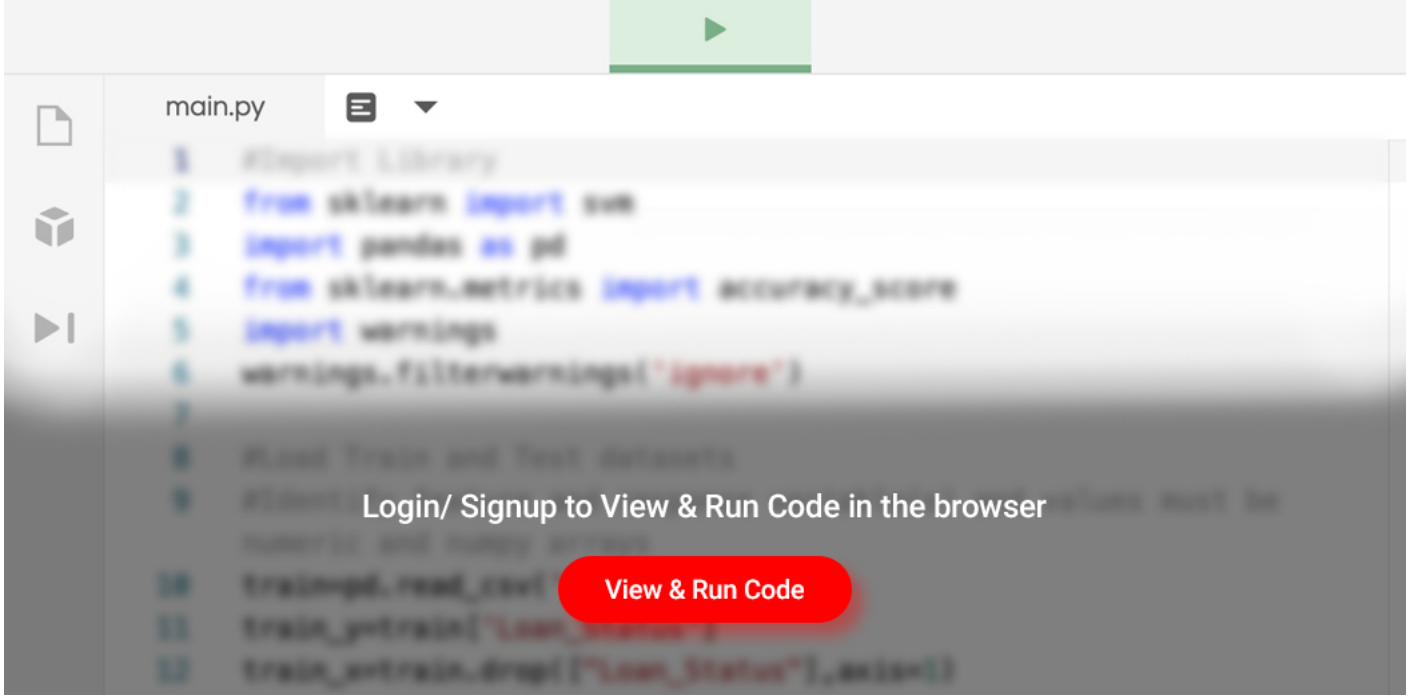
```
!wget --no-check-certificate \
    https://storage.googleapis.com/mledu-datasets/cats_and_dogs_filtered.zip \
    -O /tmp/cats_and_dogs_filtered.zip
```

view raw

get_dataset.py hosted with ❤ by GitHub

Let us also import the basic libraries. Further, I will cover future imports depending on the model including the best CNN model for image classification:

Python Code:



Preparing the Dataset

We will first prepare the dataset and separate out the images:

1. We first divide the folder contents into the train and validation directories.
2. Then, in each of the directories, create a separate directory for cats that contains only cat images, and a separate director for dogs having only dog images.

```
local_zip = '/tmp/cats_and_dogs_filtered.zip'
zip_ref = zipfile.ZipFile(local_zip, 'r')
zip_ref.extractall('/tmp')
zip_ref.close()

base_dir = '/tmp/cats_and_dogs_filtered'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')

# Directory with our training cat pictures
train_cats_dir = os.path.join(train_dir, 'cats')

# Directory with our training dog pictures
train_dogs_dir = os.path.join(train_dir, 'dogs')

# Directory with our validation cat pictures
validation_cats_dir = os.path.join(validation_dir, 'cats')

# Directory with our validation dog pictures
validation_dogs_dir = os.path.join(validation_dir, 'dogs')
```

view raw

dataset_prep.py hosted with ❤ by **GitHub**

The following code will let us check if the images have been loaded correctly:

```
# Set up matplotlib fig, and size it to fit 4x4 pics
import matplotlib.image as mpimg
nrows = 4
ncols = 4

fig = plt.gcf()
fig.set_size_inches(ncols*4, nrows*4)
pic_index = 100
train_cat_fnames = os.listdir( train_cats_dir )
train_dog_fnames = os.listdir( train_dogs_dir )

```

```
next_cat_pix = [os.path.join(train_cats_dir, fname)
                for fname in train_cat_fnames[ pic_index-8:pic_index]
               ]

next_dog_pix = [os.path.join(train_dogs_dir, fname)
                for fname in train_dog_fnames[ pic_index-8: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()
```

view raw

check_setup.py hosted with ❤ by GitHub

Now that we have our dataset ready, let us do it to the model building stage. We will be using 4 different pre-trained models on this dataset.

Examples of Pre-Trained Models for Image Classification

In this section, we cover the 4 pre-trained models for image classification as follows-

Very Deep Convolutional Networks for Large-Scale Image Recognition(VGG-16)

The VGG-16 is one of the most popular pre-trained models for image classification. Introduced in the famous ILSVRC 2014 Conference, it was and remains THE model to beat even today. Developed at the Visual Graphics Group at the University of Oxford, VGG-16 beat the then standard of AlexNet and was quickly adopted by researchers and the industry for their image Classification Tasks.

Here is the architecture of VGG-16:

Here is a more intuitive layout of the VGG-16 Model.

The following are the layers of the model:

- Convolutional Layers = 13
- Pooling Layers = 5
- Dense Layers = 3

Let us explore the layers in detail:

1. **Input**: Image of dimensions (224, 224, 3).
2. **Convolution Layer Conv1:**
    - Conv1-1: 64 filters
    - Conv1-2: 64 filters and Max Pooling
    - Image dimensions: (224, 224)
3. **Convolution layer Conv2:** Now, we increase the filters to 128
    - Input Image dimensions: (112,112)
    - Conv2-1: 128 filters
    - Conv2-2: 128 filters and Max Pooling
4. **Convolution Layer Conv3:** Again, double the filters to 256, and now add another convolution layer
    - Input Image dimensions: (56,56)
    - Conv3-1: 256 filters
    - Conv3-2: 256 filters
    - Conv3-3: 256 filters and Max Pooling
5. **Convolution Layer Conv4:** Similar to Conv3, but now with 512 filters
    - Input Image dimensions: (28, 28)
    - Conv4-1: 512 filters
    - Conv4-2: 512 filters
    - Conv4-3: 512 filters and Max Pooling
6. **Convolution Layer Conv5:** Same as Conv4
    - Input Image dimensions: (14, 14)
    - Conv5-1: 512 filters
    - Conv5-2: 512 filters
    - Conv5-3: 512 filters and Max Pooling
    - The output dimensions here are (7, 7). At this point, we flatten the output of this layer to generate a feature vector
7. **Fully Connected/Dense FC1**: 4096 nodes, generating a feature vector of size(1, 4096)
8. **Fully ConnectedDense FC2**: 4096 nodes generating a feature vector of size(1, 4096)
9. **Fully Connected /Dense FC3**: 4096 nodes, generating 1000 channels for 1000 classes. This is then passed on to a Softmax activation function

10. **Output layer**

As you can see, the model is sequential in nature and uses lots of filters. At each stage, small 3 * 3 filters are used to reduce the number of parameters all the hidden layers use the ReLU activation function. Even then, the number of parameters is **138 Billion** – which makes it a slower and much larger model to train than others.

Additionally, there are variations of the VGG16 model, which are basically, improvements to it, like VGG19 (19 layers). You can find a detailed explanation

Let us now explore how to train a VGG-16 model on our dataset-

Step 1: Image Augmentation

Since we took up a much smaller dataset of images earlier, we can make up for it by augmenting this data and increasing our dataset size. If you are working with the original larger dataset, you can skip this step and move straight on to building the model.

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

# Note that the validation data should not be augmented!
test_datagen = ImageDataGenerator( rescale = 1.0/255. )
```

view raw

img_augmentation.py hosted with ❤ by **GitHub**

Step 2: Training and Validation Sets

```
# 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 = (22

# Flow validation images in batches of 20 using test_datagen generator
validation_generator = test_datagen.flow_from_directory( validation_dir,  batch_size = 20, class_mode = 'binary', target
```

view raw

train_validation.py hosted with ❤ by **GitHub**

Step 3: Loading the Base Model

We will be using only the basic models, with changes made only to the final layer. This is because this is just a binary classification problem while these models are built to handle up to 1000 classes.

```
from tensorflow.keras.applications.vgg16 import VGG16

base_model = VGG16(input_shape = (224, 224, 3), # Shape of our images
include_top = False, # Leave out the last fully connected layer
weights = 'imagenet')
```

view raw

load_baseVGG.py hosted with ❤ by **GitHub**

Since we don't have to train all the layers, we make them non_trainable:

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

view raw

final_layer.py hosted with ❤ by **GitHub**

Step 4: Compile and Fit

We will then build the last fully-connected layer. I have just used the basic settings, but feel free to experiment with different values of dropout, and different Optimisers and activation functions.

```
# Flatten the output layer to 1 dimension
x = layers.Flatten()(base_model.output)

# Add a fully connected layer with 512 hidden units and ReLU activation
x = layers.Dense(512, activation='relu')(x)

# Add a dropout rate of 0.5
```

```
x = layers.Dropout(0.5)(x)

# Add a final sigmoid layer with 1 node for classification output
x = layers.Dense(1, activation='sigmoid')(x)

model = tf.keras.models.Model(base_model.input, x)

model.compile(optimizer = tf.keras.optimizers.RMSprop(lr=0.0001), loss = 'binary_crossentropy',metrics = ['acc'])
```

view raw

compileVGG16.py hosted with ❤ by **GitHub**

We will now build the final model based on the training and validation sets we created earlier. Please note to use the original directories itself instead of the augmented datasets I have used below. I have used just 10 epochs, but you can also increase them to get better results:

```
vgghist = model.fit(train_generator, validation_data = validation_generator, steps_per_epoch = 100, epochs = 10)
```

view raw

fitVGG16.py hosted with ❤ by **GitHub**

Awesome! As you can see, we were able to achieve a validation Accuracy of 93% with just 10 epochs and without any major changes to the model. This is where we realize how powerful transfer learning is and how useful pre-trained models for image classification can be. A caveat here though – VGG16 takes up a long time to train compared to other models and this can be a disadvantage when we are dealing with huge datasets.

That being said, I really liked how simple and intuitive this model is. Trained on the ImageNet corpus, another notable achievement of VGG-16 is that it secured the 1st Rank in the ImageNet ILSVRC-2014, and thus cemented its place in the list of top pre-trained models for image classification.

- Link to the Paper: Very Deep Convolutional Networks for Large Scale Image Recognition

Inception

While researching for this article – one thing was clear. The year 2014 has been iconic in terms of the development of really popular pre-trained models for Image Classification. While the above VGG-16 secured the 2nd rank in that years' ILSVRC, the 1st rank was secured by none other than Google – via its model GoogLeNet or Inception as it is now later called as.

The original paper proposed the Inceptionv1 Model. At only 7 million parameters, it was much smaller than the then prevalent models like VGG and AlexNet. Adding to it a lower error rate, you can see why it was a

breakthrough model. Not only this, but the major innovation in this paper was also another breakthrough – the Inception Module.

As can be seen, in simple terms, the Inception Module just performs convolutions with different filter sizes on the input, performs Max Pooling, and concatenates the result for the next Inception module. The introduction of the 1 * 1 convolution operation reduces the parameters drastically.

Though the number of layers in Inceptionv1 is 22, the massive reduction in the parameters makes it a formidable model to beat.

The Inceptionv2 model was a major improvement on the Inceptionv1 model which increased the accuracy and further made the model less complex. In the same paper as Inceptionv2, the authors introduced the Inceptionv3 model with a few more improvements on v2.

The following are the major improvements included:

- Introduction of Batch Normalisation
- More factorization
- RMSProp Optimiser

While it is not possible to provide an in-depth explanation of Inception in this article, you can go through this comprehensive article covering the Inception Model in detail: [Deep Learning in the Trenches: Understanding Inception Network from Scratch]

As you can see that the number of layers is 42, compared to VGG16's paltry 16 layers. Also, Inceptionv3 reduced the error rate to only 4.2%.

Let's see how to implement it in python-

Step 1: Data Augmentation

You will note that I am not performing extensive data augmentation. The code is the same as before. I have just changed the image dimensions for each model.

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

test_datagen = ImageDataGenerator( rescale = 1.0/255. )
```

view raw

img_augment_inception.py hosted with ❤ by **GitHub**

Step 2: Training and Validation Generators

```
train_generator = train_datagen.flow_from_directory(train_dir, batch_size = 20, class_mode = 'binary', target_size = (15
validation_generator = test_datagen.flow_from_directory(validation_dir, batch_size = 20, class_mode = 'binary', target_s
```

view raw

train_validation_Inception.py hosted with ❤ by **GitHub**

Step 3: Loading the Base Model

```
from tensorflow.keras.applications.inception_v3 import InceptionV3
base_model = InceptionV3(input_shape = (150, 150, 3), include_top = False, weights = 'imagenet')
```

view raw

loadbase_Inception.py hosted with ❤ by **GitHub**

Step 4: Compile and Fit

Just like VGG-16, we will only change the last layer.

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

view raw

final_layer.py hosted with ❤ by **GitHub**

We perform the following operations:

- Flatten the output of our base model to 1 dimension

- Add a fully connected layer with 1,024 hidden units and ReLU activation
- This time, we will go with a dropout rate of 0.2
- Add a final Fully Connected Sigmoid Layer
- We will again use RMSProp, though you can try out the Adam Optimiser too

```
from tensorflow.keras.optimizers import RMSprop

x = layers.Flatten()(base_model.output)
x = layers.Dense(1024, activation='relu')(x)
x = layers.Dropout(0.2)(x)

# Add a final sigmoid layer with 1 node for classification output
x = layers.Dense(1, activation='sigmoid')(x)

model = tf.keras.models.Model(base_model.input, x)

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

view raw

compile_Inception.py hosted with ❤ by **GitHub**

We will then fit the model:

```
inc_history = model.fit_generator(train_generator, validation_data = validation_generator, steps_per_epoch = 100, epochs
```

view raw

fit_Inception.py hosted with ❤ by **GitHub**

As a result, we can see that we get 96% Validation accuracy in 10 epochs. Also note, how this model is much faster than VGG16. Each epoch is taking around only 1/4th the time that each epoch in VGG16. Of course, you can always experiment with the different hyperparameter values and see how much better/worse it performs.

I really liked studying the Inception model. While most models at that time were merely sequential and followed the premise of the deeper and larger the model, the better it will perform- Inception and its variants broke this mold. Just like its predecessors, the Inceptionv3 achieved the top position in CVPR 2016 with only a 3.5% top-5 error rate.

Here is a link to the paper: [Rethinking the Inception Architecture for Computer Vision]

ResNet50

Just like Inceptionv3, ResNet50 is not the first model coming from the ResNet family. The original model was called the Residual net or ResNet and was another milestone in the CV domain back in 2015.

The main motivation behind this model was to avoid poor accuracy as the model went on to become deeper. Additionally, if you are familiar with Gradient Descent, you would have come across the Vanishing Gradient issue – the ResNet model aimed to tackle this issue as well. Here is the architecture of the earliest variant: ResNet34(ResNet50 also follows a similar technique with just more layers)

You can see that after starting off with a single Convolutional layer and Max Pooling, there are 4 similar layers with just varying filter sizes – all of them using 3 * 3 convolution operation. Also, after every 2 convolutions, we are bypassing/skipping the layer in-between. This is the main concept behind ResNet

models. These skipped connections are called ‘identity shortcut connections” and uses what is called residual blocks:

In simple terms, the authors of the ResNet propose that fitting a residual mapping is much easier than fitting the actual mapping and thus apply it in all the layers. Another interesting point to note is the authors of ResNet are of the opinion that the more layers we stack, the model should not perform worse.

This is contrary to what we saw in Inception and is almost similar to VGG16 in the sense that it is just stacking layers on top of the other. ResNet just changes the underlying mapping.

The ResNet model has many variants, of which the latest is ResNet152. The following is the architecture of the ResNet family in terms of the layers used:

Let us now use ResNet50 on our dataset:

Step 1: Data Augmentation and Generators

```
# Add our data-augmentation parameters to ImageDataGenerator

train_datagen = ImageDataGenerator(rescale = 1./255., rotation_range = 40, width_shift_range = 0.2, height_shift_range =

test_datagen = ImageDataGenerator(rescale = 1.0/255.)

train_generator = train_datagen.flow_from_directory(train_dir, batch_size = 20, class_mode = 'binary', target_size = (22

validation_generator = test_datagen.flow_from_directory( validation_dir, batch_size = 20, class_mode = 'binary', target_
```

view raw

augment_ResNet.py hosted with ❤ by GitHub

Step 2: Import the base model

```
from tensorflow.keras.applications import ResNet50

```

```
base_model = ResNet50(input_shape=(224, 224,3), include_top=False, weights="imagenet")
```

view raw

load_base_Resnet.py hosted with ❤ by GitHub

Again, we are using only the basic ResNet model, so we will keep the layers frozen and only modify the last layer:

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

view raw

final_layer.py hosted with ❤ by GitHub

Step 3: Build and Compile the Model

Here, I would like to show you an even shorter code for using the ResNet50 model. We will use this model just as a layer in a Sequential model, and just add a single Fully Connected Layer on top of it.

```
from tensorflow.keras.applications import ResNet50
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import Dense, Flatten, GlobalAveragePooling2D

base_model = Sequential()
base_model.add(ResNet50(include_top=False, weights='imagenet', pooling='max'))
base_model.add(Dense(1, activation='sigmoid'))
```

view raw

compile1_Resnet.py hosted with ❤ by GitHub

We compile the model and this time let us try the SGD optimizer:

```
base_model.compile(optimizer = tf.keras.optimizers.SGD(lr=0.0001), loss = 'binary_crossentropy', metrics = ['acc'])
```

view raw

compile2_Resnet.py hosted with ❤ by GitHub

Step 4: Fitting the model

```
resnet_history = base_model.fit(train_generator, validation_data = validation_generator, steps_per_epoch = 100, epochs =
```

view raw

fit_ResNet.py hosted with ❤ by GitHub

The following is the result we get-

You can see how well it performs on our dataset and this makes ResNet50 one of the most widely used Pre-trained models. Just like VGG, it also has other variations as we saw in the table above. Remarkably, ResNet not only has its own variants, but it also spawned a series of architectures based on ResNet. These include ResNeXt, ResNet as an Ensemble, etc. Additionally, the ResNet50 is among the most popular models out there and achieved a top-5 error rate of around 5%

The following is the link to the paper: Deep Residual Learning for Image Recognition

EfficientNet

We finally come to the latest model amongst these 4 that have caused waves in this domain and of course, it is from Google. In EfficientNet, the authors propose a new Scaling method called **Compound Scaling.** The long and short of it is this: The earlier models like ResNet follow the conventional approach of scaling the dimensions arbitrarily and by adding up more and more layers.

However, the paper proposes that if we scale the dimensions by a fixed amount at the same time and do so uniformly, we achieve much better performance. The scaling coefficients can be in fact decided by the user.

Though this scaling technique can be used for any CNN-based model, the authors started off with their own baseline model called EfficientNetB0:

MBConv stands for mobile inverted bottleneck Convolution(similar to MobileNetv2). They also propose the Compound Scaling formula with the following scaling coefficients:

- Depth = 1.20
- Width = 1.10
- Resolution = 1.15

This formula is used to again build a family of EfficientNets – EfficientNetB0 to EfficientNetB7. The following is a simple graph showing the comparative performance of this family vis-a-vis other popular models:

As you can see, even the baseline B0 model starts at a much higher accuracy, which only goes on increasing, and that too with fewer parameters. For instance, EfficientB0 has only 5.3 million parameters!

The simplest way to implement EfficientNet is to install it and the rest of the steps are similar to what we have seen above.

Installing EfficientNet:

```
!pip install -U efficientnet
```

Import it

```
import efficientnet.keras as efn
```

view raw

import_Efficientnet.py hosted with ❤ by GitHub

Step 1: Image Augmentation

We will use the same image dimensions that we used for VGG16 and ResNet50. By now, you would be familiar with the Augmentation process:

```
# Add our data-augmentation parameters to ImageDataGenerator

train_datagen = ImageDataGenerator(rescale = 1./255., rotation_range = 40, width_shift_range = 0.2, height_shift_range =

test_datagen = ImageDataGenerator(rescale = 1.0/255.)

train_generator = train_datagen.flow_from_directory(train_dir, batch_size = 20, class_mode = 'binary', target_size = (22

validation_generator = test_datagen.flow_from_directory( validation_dir, batch_size = 20, class_mode = 'binary', target_
```

view raw

augment_ResNet.py hosted with ❤ by GitHub

Step 2: Loading the Base Model

We will be using the B0 version of EfficientNet since it is the simplest of the 8. I urge you to experiment with the rest of the models, though do keep in mind that the models go on becoming more and more complex, which might not be the best suited for a simple binary classification task.

```
base_model = efn.EfficientNetB0(input_shape = (224, 224, 3), include_top = False, weights = 'imagenet')
```

view raw

load_base_EfficientNet.py hosted with ❤ by GitHub

Again, let us freeze the layers:

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

view raw

final_layer.py hosted with ❤ by GitHub

Step 3: Build the model

Just like Inceptionv3, we will perform these steps at the final layer:

```
x = base_model.output
x = Flatten()(x)
x = Dense(1024, activation="relu")(x)
x = Dropout(0.5)(x)

# Add a final sigmoid layer with 1 node for classification output
predictions = Dense(1, activation="sigmoid")(x)
model_final = Model(input = base_model.input, output = predictions)
```

view raw

build_EfficientNet.py hosted with ❤ by GitHub

Step 4: Compile and Fit

Let us again use the RMSProp Optimiser, though here, I have introduced a decay parameter:

```
model_final.compile(optimizers.rmsprop(lr=0.0001, decay=1e-6),loss='binary_crossentropy',metrics=['accuracy'])
```

view raw

compile_EfficientNet.py hosted with ❤ by **GitHub**

We finally fit the model on our data:

```
eff_history = model_final.fit_generator(train_generator, validation_data = validation_generator, steps_per_epoch = 100,
```

view raw

fit_EfficientNet.py hosted with ❤ by **GitHub**

There we go – we got a whopping 98% accuracy on our validation set in only 10 epochs. I urge you to try training the larger dataset with EfficientNetB7 and share the results with us below.

The following is the link to the paper: [EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks]

Conclusion

To summarize, in this article, I introduced to you 4 of the top State-of-the-Art pre-trained models for image classification. Here is a handy table for you to refer these models and their performance:

I have only provided an overview of the top 4 pre-trained models for image classification and how to implement them. However, this is a continuously growing domain and there is always a new model to look forward to and push the boundaries further. I cannot wait to explore these new models and I also urge you to try out the above models on different datasets with different parameters, and share your results with us in the comments below!

Frequently Asked Questions

Q1. What is VGG-16 network?

A. VGG-16 is a convolutional neural network that has 16 layers in it. These networks can be trained on more than a million images to classify images into over 1000 categories.

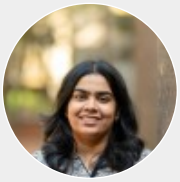
Q2. What is ResNet50?

A. ResNet-50 is a convolutional neural network that has 50 layers in it. It consists of 48 convolutional layers, one MaxPool layer, and one average pool layer.

Q3. What is EfficientNet in CNN?

A. EfficientNet is a CNN scaling method that uniformly scales all the dimensions of an image (depth, width, and resolution) using a compound coefficient.

Article Url - https://www.analyticsvidhya.com/blog/2020/08/top-4-pre-trained-models-for-image-classification-with-python-code/



Purva Huilgol