Transfer learning and Image classification using Keras on Kaggle kernels.

Rising Odegua
Nov 2, 2018 · 11 min read

In my last **post**, we trained a **convnet** to differentiate dogs from cats. We trained the convnet from scratch and got an accuracy of about 80%. Not bad for a model trained on very little dataset (4000 images).

But in real world/production scenarios, our model is actually under-performing.

Although we suggested tuning some **hyperparameters** — epochs, learning rates, input size, network depth, backpropagation algorithms e.t.c — to see if we could increase our accuracy.

Well, I did try...

And truth is, after tuning, re-tuning, not-tuning , my accuracy wouldn't go above 90% and at a point It was useless.

> *Of course having more data would have helped our model; But remember we're working with a small dataset, a common problem in the field of deep learning.*

But alas! there is a way….

Image from pixabay

Transfer learning walks in….

But then you ask, what is Transfer learning?

Well, TL (Transfer learning) is a popular training technique used in deep learning; where models that have been trained for a task are reused as base/starting point for another model.

To train an Image classifier that will achieve near or above human level accuracy on Image classification, we'll need massive amount of data, large compute power, and lots of time on our hands. This I'm sure most of us don't have.

Knowing this would be a problem for people with little or no resources, some smart researchers built models, trained on large image datasets like **ImageNet**, **COCO**, **Open Images**, and decided to share their models to the general public for reuse.

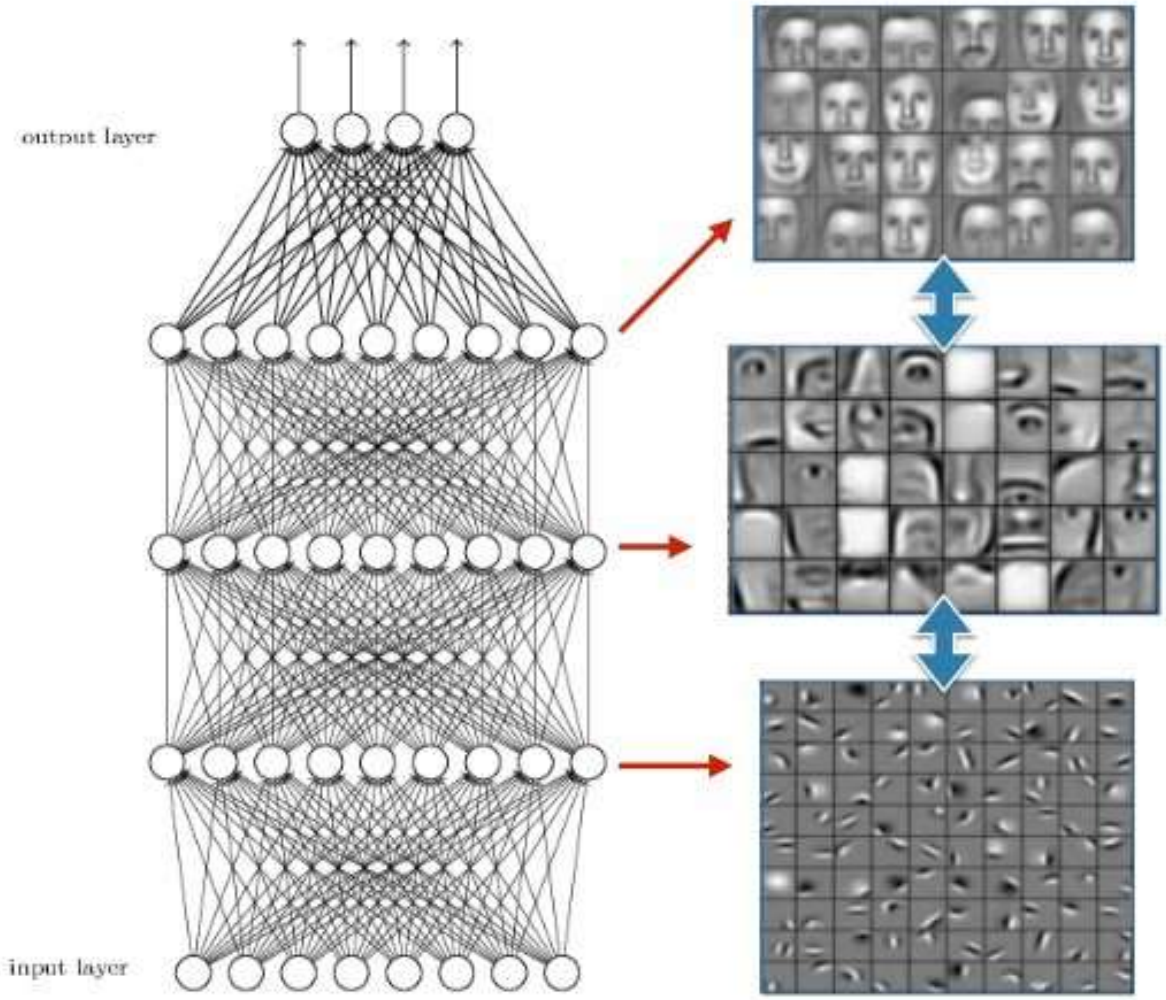
Image from **Marvel comics**

This means you should never have to train an Image classifier from scratch again, unless you have a very, very large dataset different from the ones above or you want to be an **hero or thanos.**

I know you have questions, like…

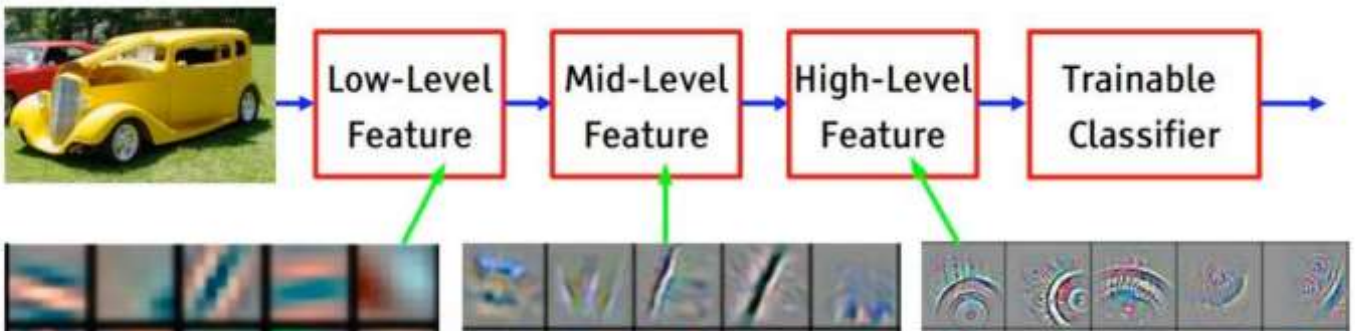
- Why does transfer learning work?

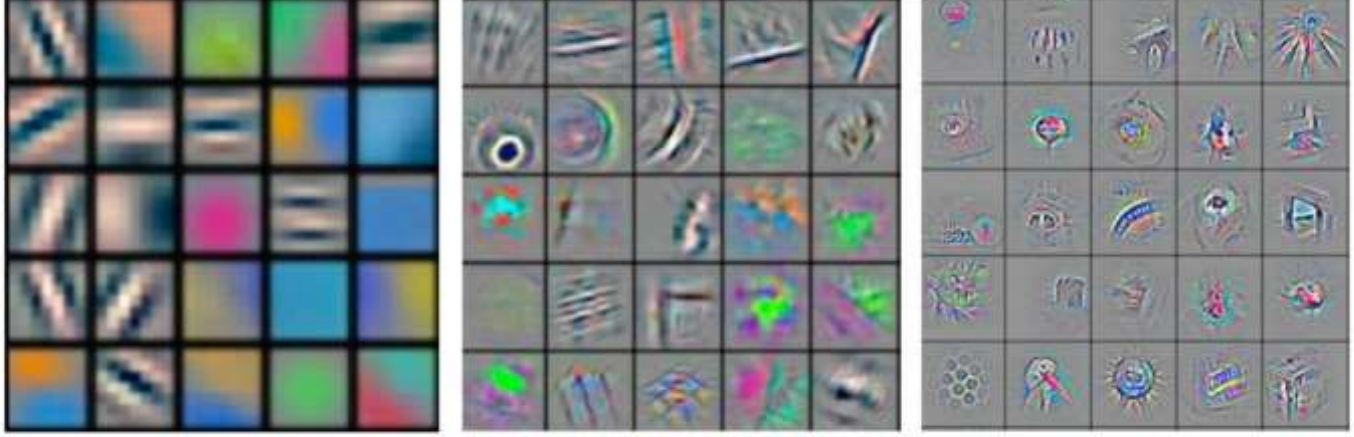
Well Transfer learning works for Image classification problems because Neural Networks learn in an increasingly complex way. i.e The deeper you go down the network the more image specific features are learnt.



A neural network learns to detect objects in increasing level of complexity | Image source: cnnetss.com

Let's build some intuition to understand this better. In a neural network trying to detect faces,we notice that the network learns to detect edges in the first layer, some basic shapes in the second and complex features as it goes deeper.





Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

So the idea here is that all Images have shapes and edges and we can only identify differences between them when we start extracting higher level features like-say nose in a face or tires in a car. Only then can we say, okay; this is a person, because it has a nose and this is an automobile because it has a tires.

The take-away here is that the earlier layers of a neural network will always detect the same basic shapes and edges that are present in both the picture of a car and a person.

Now, taking this intuition to our problem of differentiating **dogs from cats**, it means we can use models that have been trained on huge dataset containing different types of animals.
This works because these models have learnt already the basic shape and structure of animals and therefore all we need to do, is teach it (model) the high level features of our new images.

All I'm trying to say is that we need a network already trained on a large image dataset like **ImageNet** (contains about 1.4 million labeled images and 1000 different categories including animals and everyday objects).
Since this model already knows how classify different animals, then we can use this existing knowledge to quickly train a new classifier to identify our specific classes (cats and dogs).

> I mean a person who can boil eggs should know how to boil just water right?

Now that we have an understanding/intuition of what Transfer Learning is, let's talk about **pretrained** networks.

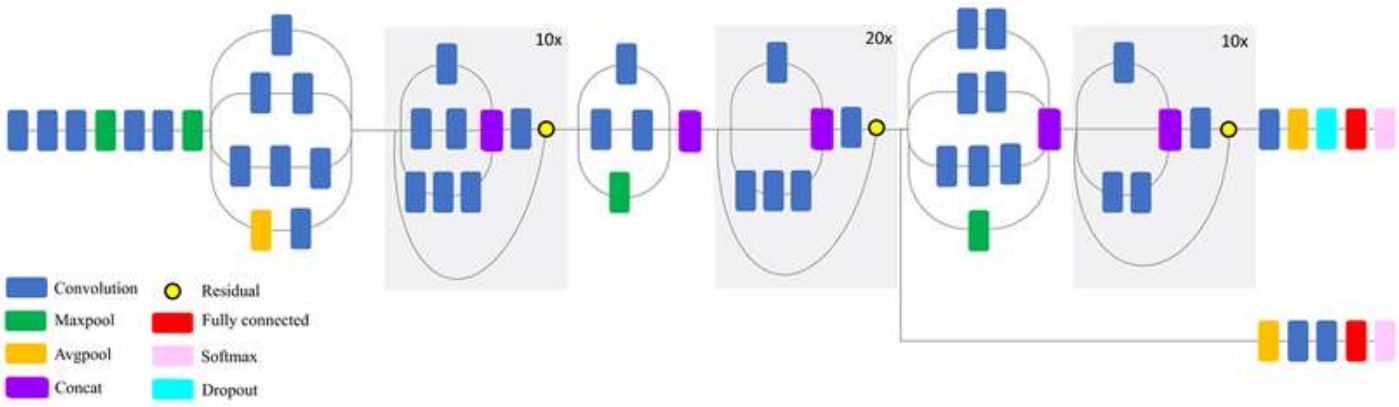
There are different variants of **pretrained** networks each with its own architecture, speed, size, advantages and disadvantages.

Keras comes prepackaged with many types of these **pretrained** models. Some of them are:

- **VGGNET :** Introduced by Simonyan and Zisserman in their 2014 paper, *Very Deep Convolutional Networks for Large Scale Image Recognition*.
- **RESNET :** First introduced by He et al. in their 2015 paper, *Deep Residual Learning for Image Recognition*
- **INCEPTION**: The "Inception" micro-architecture was first introduced by Szegedy et al. in their 2014 paper, *Going Deeper with Convolutions*:
- **XCEPTION:** Xception was proposed by François Chollet , the creator of the **Keras** library.

and many more. Detailed explanation of some of these architectures can be found **here.**

We'll be using the **InceptionResNetV2** in this tutorial, feel free to try other models.



InceptionResNetV2 model | Image source: researchgate

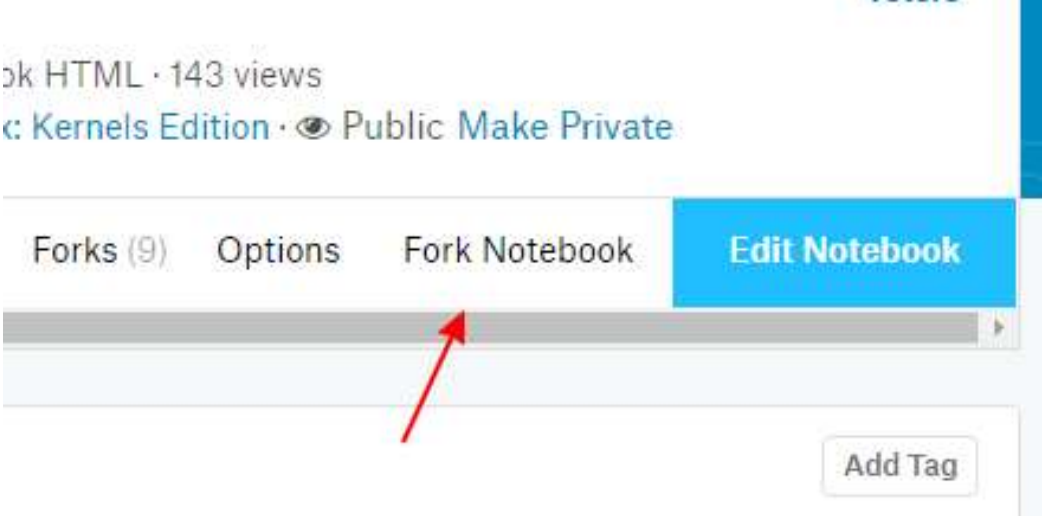
The **InceptionResNetV2** is a recent architecture from the **INCEPTION** family. It works really well and is super fast for many reasons, but for the sake of brevity, we'll leave the details and stick to just using it in this post.

If you're interested in the details of how the **INCEPTION** model works then go **here**.

. . .

With the not-so-brief introduction out of the way, let's get down to actual coding.

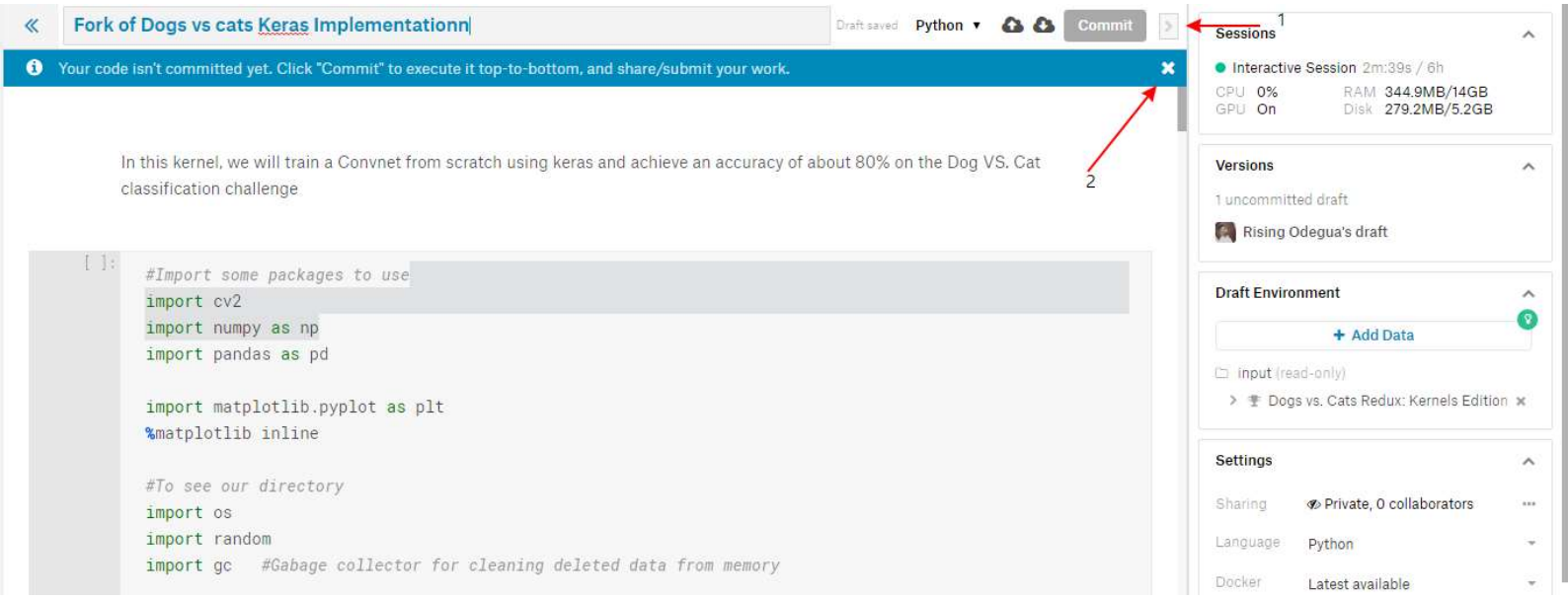
We'll be using almost the same code from our first Notebook, the difference will be pretty simple and straightforward, as Keras makes it easy to call pretrained model.



Fork your previous notebook on kaggle

If you followed my previous **post** and already have a kernel on **kaggle**, then simply fork your Notebook to create a new version. We'll be editing this version.

A fork of your previous notebook is created for you as shown below



The new version opens like this.

1. Close the settings bar, since our GPU is already activated. Please confirm your GPU is on as it could greatly impact training time.
2. Cancel the commit message. Do not commit your work yet, as we're yet to make any change.

Now, run the code blocks from the start one after the other until you get to the cell where we created our **Keras** model, as shown below.

```
from keras import layers
from keras import models
from keras import optimizers
from keras.preprocessing.image import ImageDataGenerator
from keras.preprocessing.image import img_to_array, load_img

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5))  #Dropout for regularization
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))  #Sigmoid function at the end because we have just two classes
```

Keras model block

Click the **+ button with an arrow pointing up** to create a new code cell on top of this current one.

```
[ ]:

[ ]:  from keras import layers
      from keras import models
```

empty code cell created

Now, let's call our pretrained model...

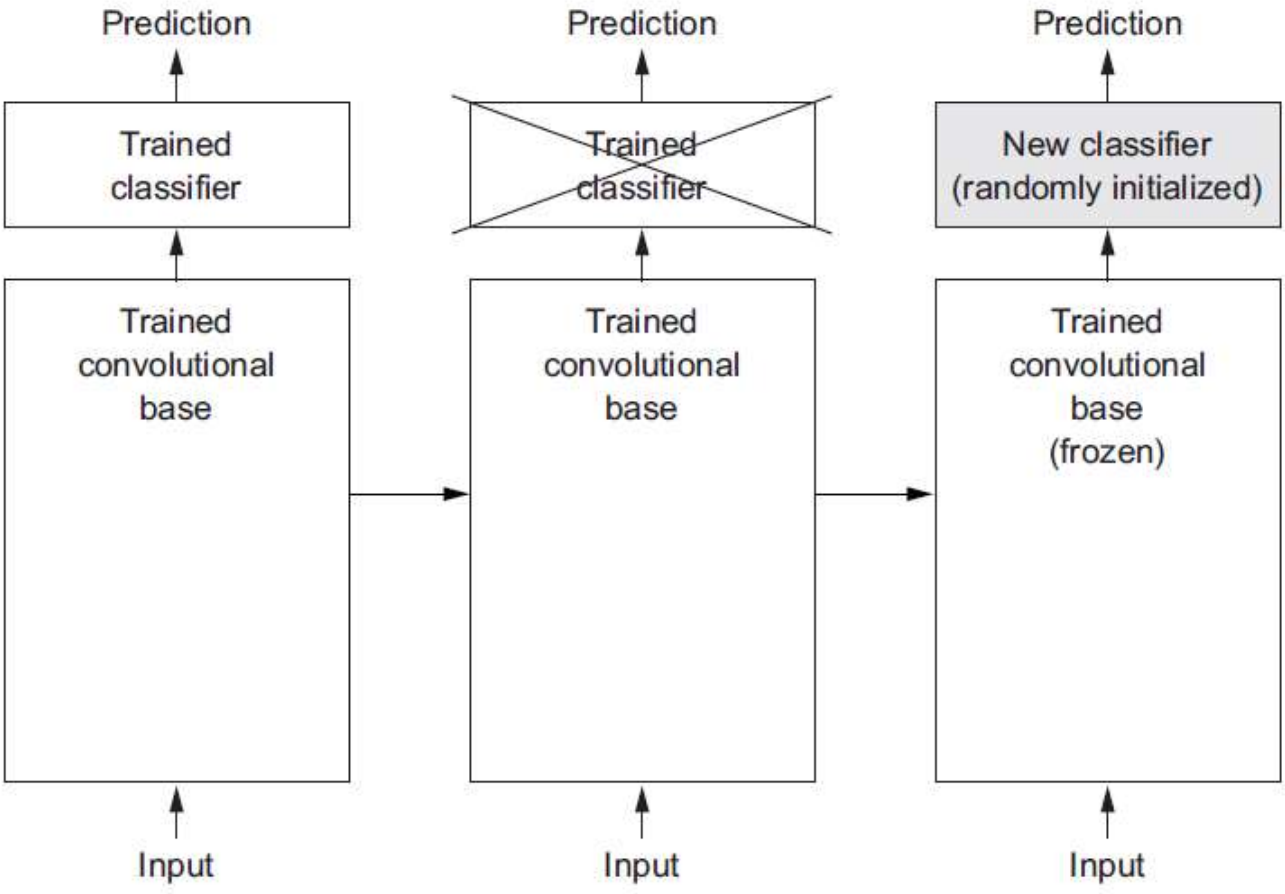
```
12]:  from keras.applications import InceptionResNetV2
      conv_base = InceptionResNetV2(weights='imagenet', include_top=False, input_shape=(150,150,3))
```

1. Here, we import the **InceptionResNetV2** model.

2. Here, we tell **keras** to download the model's **pretrained** weights and save it in the variable **conv_base.**
   To configure what we actually download, we pass in some important parameters such as:

3. *weights* [**imagenet**]: We tell keras to fetch InceptionReNetV2 that was trained on the **imagenet** dataset.

4. *include_top* [**False**]: This tells **Keras** not to download the fully connected layers of the pretrained model. This is because the top layer (fully connected layers) does the final classification.
   I.e After the convolution layers extract basic features such as edges, blobs or lines from the input images, the fully connected layer then classifies them into categories.
   Since all we need is 2 class (dogs and cats) classifier, we are going to remove the former and add our own.



Removing the top classifier and adding our own. Image source: Deep Learning with Python by Francois Chollet

5. Here we specify our input dimension.

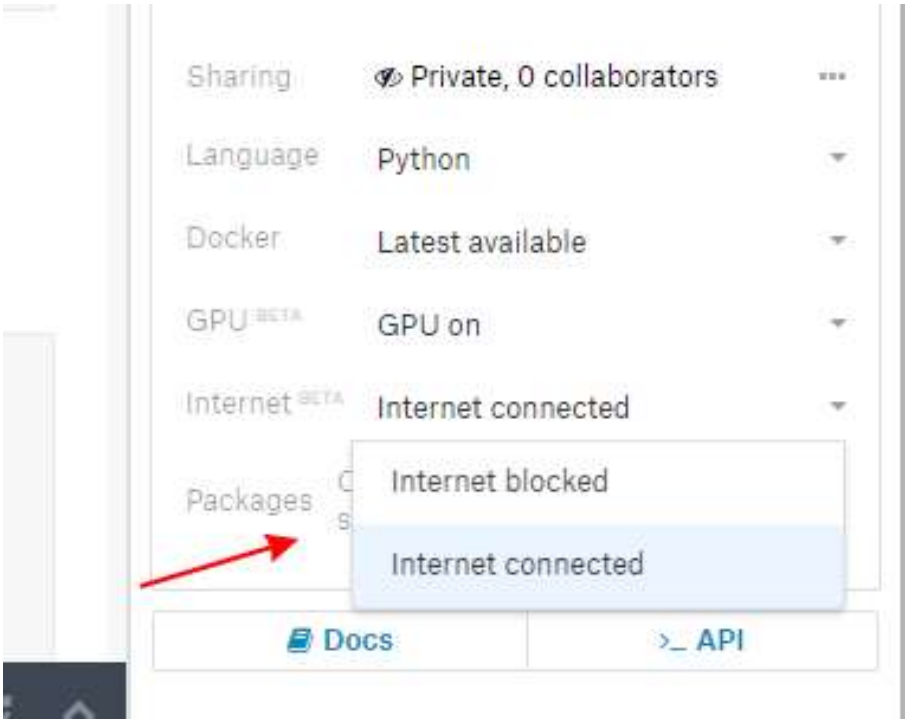
. . .

Click **shift+Enter** to run the code block.

```
Using TensorFlow backend.

Downloading data from https://github.com/fchollet/deep-learning-models/releases/download/v0.7/inception_resnet_v2_weights_tf_dim_ordering_tf_
kernels_notop.h5


---------------------------------------------------------------------------
gaierror                                  Traceback (most recent call last)
/opt/conda/lib/python3.6/urllib/request.py in do_open(self, http_class, req, **http_conn_args)
   1317                 h.request(req.get_method(), req.selector, req.data, headers,
-> 1318                           encode_chunked=req.has_header('Transfer-encoding'))
   1319             except OSError as err: # timeout error

/opt/conda/lib/python3.6/http/client.py in request(self, method, url, body, headers, encode_chunked)
```

This error appears when your internet on kaggle is blocked.

If you get this error when you run the code, then your internet access on Kaggle kernels is blocked.



To activate it, open your settings menu, scroll down and click on **internet** and select ***Internet connected.***

Your kernel automatically refreshes. So you have to run every cell from the top again, until you get to the current cell.

Rerunning the code downloads the pretrained model from the **keras** repository on **github**.

```
Using TensorFlow backend.

Downloading data from https://github.com/fchollet/deep-learning-models/releases/download/
ernels_notop.h5
219062272/219055592 [==============================] - 2s 0us/step
```

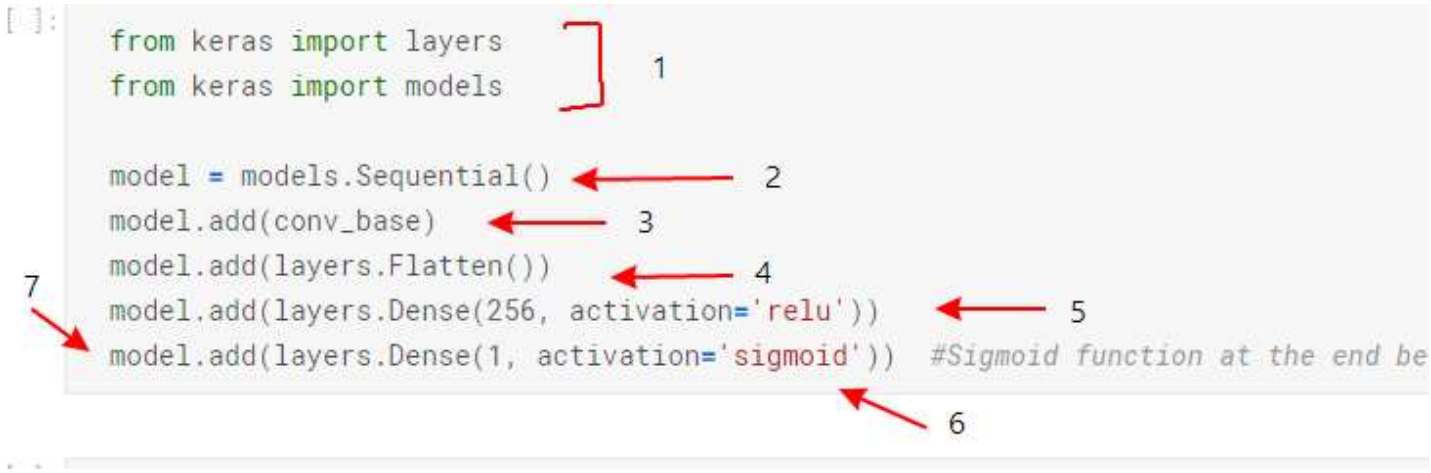
Downloading our pretrained model from github

```
conv_base.summary()
                                                                 block8_10_conv[0][0]
_________________________________________________________________________________________
conv_7b (Conv2D)             (None, 3, 3, 1536)   3194880     block8_10[0][0]
_________________________________________________________________________________________
conv_7b_bn (BatchNormalization) (None, 3, 3, 1536)   4608        conv_7b[0][0]
_________________________________________________________________________________________
conv_7b_ac (Activation)      (None, 3, 3, 1536)   0           conv_7b_bn[0][0]
=========================================================================================


Total params: 54,336,736
Trainable params: 54,276,192
Non-trainable params: 60,544
_________________________________________________________________________________________
```

We can call the ***.summary( )*** function on the model we downloaded to see its architecture and number of parameters.
You notice a whooping 54 million plus parameters. This is massive and we definitely can not train it from scratch. But thanks to Transfer learning we can simply re-use it without training.

Next, we create our fully connected layers (classifier) which we add on-top of the model we downloaded. This is the classifier we are going to train. I.e after

connecting the **InceptionResNetV2** to our classifier, we will tell keras to train only our classifier and freeze the **InceptionResNetV2** model.



1. We import layers and models from keras.
2. We create a Sequential model.
3. **IMPORTANT!** don't worry, I'm not screaming… here we add our **conv_base (**InceptionResNetV2) to the model.
4. Here, we flatten the output from the **conv_base** because we want to pass it to our fully connected layer (classifier).
5. To keep things simple, I added a **Dense** network with an output of 256 (**number not fixed**) and used the popular **ReLU** activation we talked about in my last **post.**
6. We use a **sigmoid** for our final layer like we did last time.
7. The last layer has just 1 output. (Probability of classes)
   See my last **post** for more clarifications.

Hold Shift+Enter to run your code.

• • •

Next, let's preview our architecture:

```
#Lets see our model
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
inception_resnet_v2 (Model)  (None, 3, 3, 1536)        54336736
_________________________________________________________________
flatten_1 (Flatten)          (None, 13824)             0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               3539200
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 257
=================================================================
Total params: 57,876,193
Trainable params: 57,815,649
Non-trainable params: 60,544
_________________________________________________________________
```

Full model summary

We can see that our parameters has increased from roughly 54 million to almost 58 million, meaning our classifier has about 3 million parameters.

Now we're going freeze the **conv_base** and train only our own.

```
print('Number of trainable weights before freezing the conv base:', len(model.trainable_weights))
conv_base.trainable = False
print('Number of trainable weights after freezing the conv base:', len(model.trainable_weights))
```

```
Number of trainable weights before freezing the conv base: 492
Number of trainable weights after freezing the conv base: 4
```

Freeze!

1. We print the number of weights in the model before freezing the **conv_base** (*492*).

2. Here, we set the ***trainable*** property of the **conv_base** layer to False (we don't want to train you).

3. Print the number of weights after freezing the **conv_base** (*down to just 4*)**.**

Almost done, just some minor changes and we can start training our model.

First little change is to increase our learning rate slightly from **0.0001** (*1e-5*) in our last model to **0.0002**(*2e-5*)**.** I decided to use **0.0002** after some experimentation and it kinda worked better**.** (you can *do* some more tuning here)

```
#We'll use the RMSprop optimizer with a learning rate of 0.0001
#We'll use binary_crossentropy loss because its a binary classification
from keras import optimizers

model.compile(loss='binary_crossentropy', optimizer=optimizers.RMSprop(lr=2e-5), metrics=['acc'])
```

Increase the learning rate slightly

Next, run all the cells below the **model.*compile*** block until you get to the cell where we called **fit** on our model**.** Here we'll change one last parameter which is the **epoch** size**.**

```
]:
    #The training part
    #We train for 64 epochs with about 100 steps per epoch
    history = model.fit_generator(train_generator,
                                  steps_per_epoch=ntrain // batch_size,
                                  epochs=20,
                                  validation_data=val_generator,
                                  validation_steps=nval // batch_size)
```

We reduce the epoch size to **20.** The reason for this will be clearer when we plot accuracy and loss graphs later.
Note: I decided to use 20 after trying different numbers. This is what we call ***Hyperparameter*** tuning in deep learning.

Finally, time to start training.

Run your code and go grab a cup of water. No not coffee! Water, cos water is life.

. . .

Well, before I could get some water, my model finished training. So let's evaluate its performance.

> Picture showing the power of Transfer Learning.

```
100/100 [==============================] - 27s 274ms/step - loss: 0.3477 - acc: 0.8453 - val_loss: 0.3043 - val_acc: 0.9525
Epoch 14/20
100/100 [==============================] - 28s 278ms/step - loss: 0.3270 - acc: 0.8572 - val_loss: 0.3616 - val_acc: 0.9550
Epoch 15/20
100/100 [==============================] - 28s 279ms/step - loss: 0.3203 - acc: 0.8569 - val_loss: 0.4336 - val_acc: 0.9337
Epoch 16/20
100/100 [==============================] - 28s 279ms/step - loss: 0.3144 - acc: 0.8606 - val_loss: 0.3774 - val_acc: 0.9425
Epoch 17/20
```

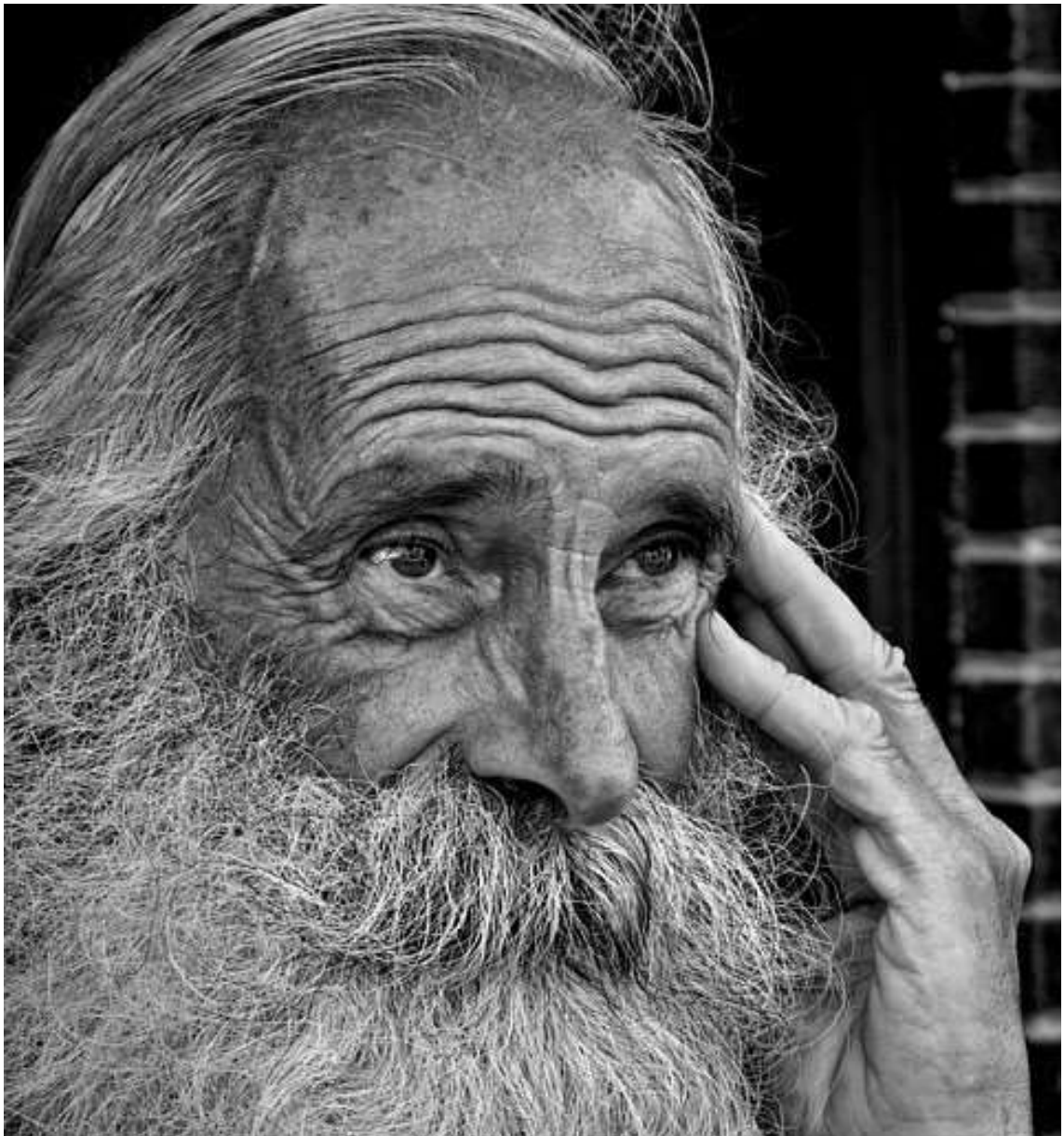
```
100/100 [==============================] - 28s 279ms/step - loss: 0.3062 - acc: 0.8713 - val_loss: 0.4489 - val_acc: 0.9325
Epoch 18/20
100/100 [==============================] - 28s 277ms/step - loss: 0.3153 - acc: 0.8531 - val_loss: 0.5416 - val_acc: 0.9100
Epoch 19/20
100/100 [==============================] - 28s 281ms/step - loss: 0.3051 - acc: 0.8659 - val_loss: 0.3540 - val_acc: 0.9500
Epoch 20/20
100/100 [==============================] - 28s 280ms/step - loss: 0.3161 - acc: 0.8563 - val_loss: 0.2972 - val_acc: 0.9563
```

We reach an accuracy of about 96% in just 20 epochs

We clearly see that we have achieved an accuracy of about 96% in just 20 epochs. Super fast and accurate.

If the ***dogs vs cats*** competition weren't closed and we made predictions with this model, we would definitely be among the top if not the first.
And remember, we used just 4000 images from a total of about 25,000.

What happens when we use all 25000 images for training combined with the technique (Transfer learning) we just learnt?



A wise man thinking... Image source **Pixabay**

> *Well, a very wise scientist once said…*

> A not-too-fancy algorithm with enough data would certainly do better than a fancy algorithm with little data.

And it has been proven true!

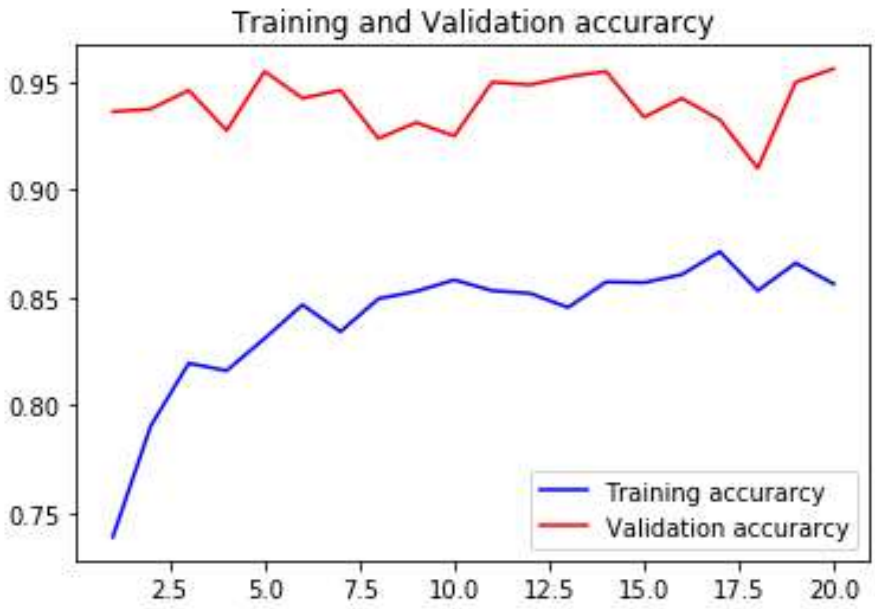
Okay, we've been talking numbers for a while now, let's see some visuals…

```
#get the details form the history object
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

#Train and validation accuracy
plt.plot(epochs, acc, 'b', label='Training accurarcy')
plt.plot(epochs, val_acc, 'r', label='Validation accurarcy')
plt.title('Training and Validation accurarcy')
plt.legend()

plt.figure()
#Train and validation loss
plt.plot(epochs, loss, 'b', label='Training loss')
plt.plot(epochs, val_loss, 'r', label='Validation loss')
```

```
plt.title('Training and Validation loss')
plt.legend()

plt.show()
```

Without changing your plotting code, run the cell block to make some accuracy and loss plots.

And we get these plots below…



So what can we read of this plot?Well, we can clearly see that our validation accuracy starts doing well even from the beginning and then plateaus out after just a few epochs.
Now you know why I decreased my epoch size from 64 to 20.

Finally, let's see some predictions. We are going to use the same prediction code. Just run the code block.

> After running mine, I get the prediction for 10 images as shown below...

Predictions for ten Images from our test set

And our classifier got a 10 out of 10. Pretty nice and easy right?

Well, This is it. This is where I stop typing and leave you to go harness the power of Transfer learning.

So, Happy coding…

Link to this notebook on Kaggle**.**

Link to this notebooks on Github**.**

Some amazing post and write-ups I referenced.

- CS231n Convolutional Neural Networks for Visual Recognition.
- Deep learning with python by Francois Chollet the creator of Keras.
- Covolution Neural network basics.
- A great medium post on Transfer learning.
- Another great medium post on Inception models.
- A gentle approach to Transfer learning.

> Questions, comments and contributions are always welcome.

*Connect with me on **twitter**.*

*Connect with me on **Instagram.***

Machine Learning | Keras | Transfer Learning | Inception Model | Image Classification