**Transfer Learning**

While developing deep convolutional neural network architectures may take days or even weeks to train on sizeable datasets. The computation cost time while training can be improved by reusing the weights from pre-trained architectures developed for standard computer vision benchmark datasets, like the ImageNet dataset. According to the need of problem statement, various pre-trained architectures can be downloaded and integrated into a new architecture. Here, you will understand to use pre-trained CNN architectures to build a Convolutional Neural Network (CNN). So the method of reusing a previously learned model on a new problem is known as Transfer Learning.

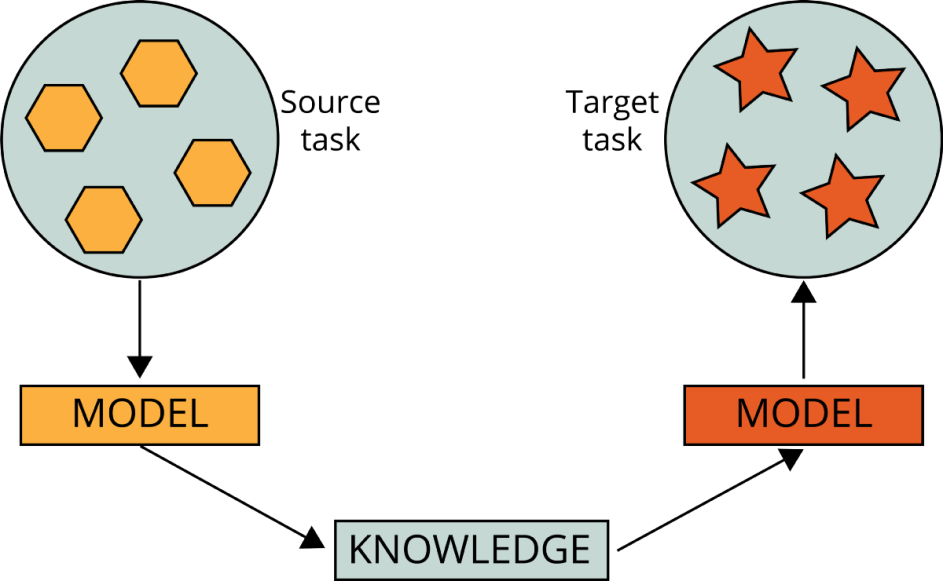
**What Is Transfer Learning?**

Transfer learning is a research problem in machine learning that focuses on storing knowledge gained while solving one problem and applying it to a different but related problem. For example, knowledge gained while learning to recognize cats can apply while recognizing new cat images. It has the advantage of reducing the training time for a CNN and can result in a lower error. The weights in reused layers may be utilized as the initial point for the training procedure and adjusted in the new problem statement. This way, you can use transfer learning as a weight initialization method. So, now let's see how to use the concept of Transfer Learning to apply to a dataset.

## What is Pre-Trained Models?

A pre-trained model (VGG, Inception, MobileNet) is a model that was trained on the ImageNet dataset to solve a problem equivalent to the one that you want to solve. Hence, due to the computational cost and time of training such models, people usually import and use the pre-trained models.

But despite the advantages of using the pre-trained models, you need to be careful when to use them. Transfer Learning won't work when there is a mismatch between the dataset used while training those models and your dataset. The pre-trained models may converge but around a local minima. Thus, the result after training won't be a good one.



**Some Popular Pre-trained Models**

**VGG-16**

VGG-16, a CNN architecture that was the runner-up in the ILSVR(Imagenet) competition in 2014. VGG-16 concentrated on maintaining convolution layers of a 3x3 filter with a stride of 1, same padding, and max pool layer of 2x2 filter of stride 2, followed by a 2 FC(fully connected layers) followed by a softmax for output. It follows the sequence of convolution and max pool layers throughout the architecture. The 16 in VGG16 refers to it having 16 layers that have weights. This network is pretty large and has about 138 million (approx) parameters.

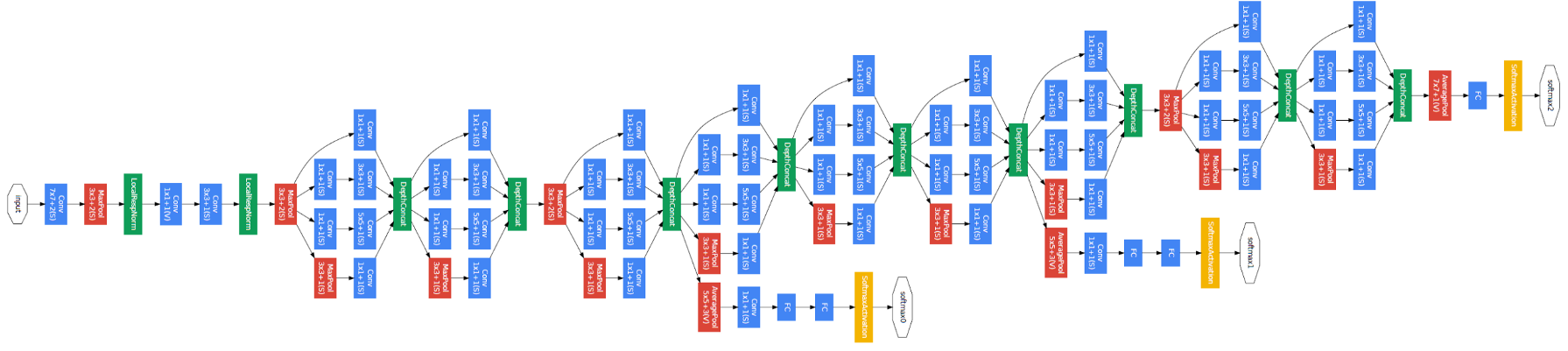


**Inception**

Inception is a CNN architecture that won the ILSVR(Imagenet) competition in 2014. It is a combination of 1×1, 3×3, and 5×5 Convolutional layers concatenated into a single output vector forming the input of the next stage. With the layers mentioned above, there are two main add-ons in the initial inception layer:

* 1×1 Convolutional layer before applying another layer, used for dimensionality reduction.
* A parallel Max Pooling layer.

After all the Convolution layers, it has three dense layers (2 fully connected layers; 1 layer for output).



**Xception**

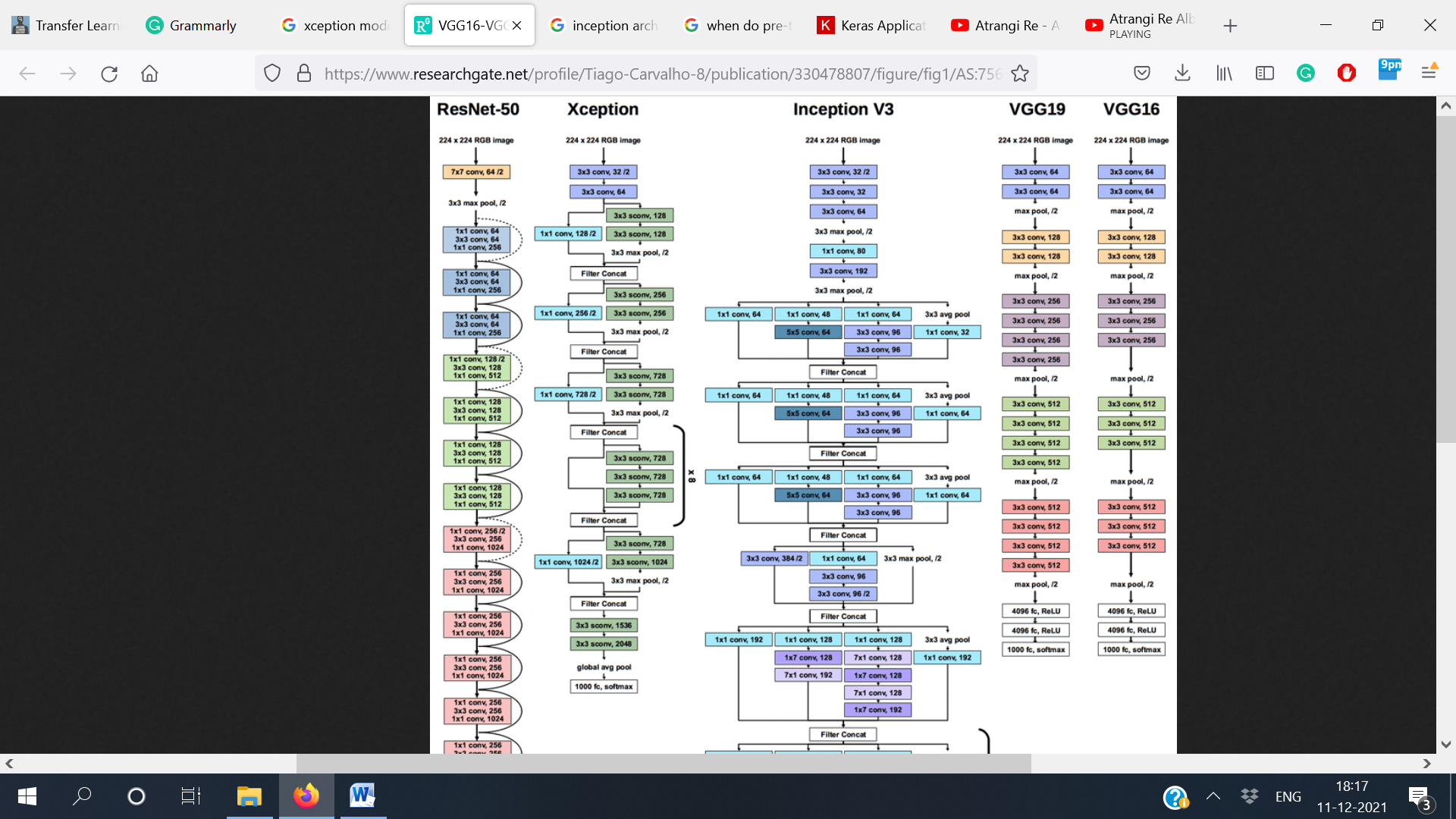
Xception is a CNN architecture that won the ILSVR(Imagenet) competition in 2017. With a modified depthwise separable convolution, it is even better than Inception. It has provided good results on various datasets, and many consider it one of the best CNN architecture.

The actual depthwise separable convolution is the depthwise convolution followed by a pointwise convolution.

1. Depthwise convolution is the channel-wise n×n spatial convolution. For example, if you have 3-channels, then will be 5 'n×n' spatial convolutions.
2. Pointwise convolution is the 1×1 convolution to adjust the dimension.

The advantage of using Depthwise Separable Convolution is that it reduces the number of parameters, and hence the model is lighter.

Now the Xceptionarchitecture uses a modified form of depthwise Separable Convolution. The modified depthwise separable convolution is the pointwise convolution followed by a depthwise convolution. This change is motivated by the inception module in Inception that 1×1 convolution is done first before any n×n spatial convolutions. Thus, it is a bit different from the original one.



**MobileNet**

As the name applied, the MobileNet model is designed for mobile applications, and it is the first mobile model using TensorFlow. MobileNet uses depthwise separable convolutions. It significantly reduces the number of parameters when compared to the network with regular convolutions with the same depth in the nets. This results in lightweight deep neural networks.

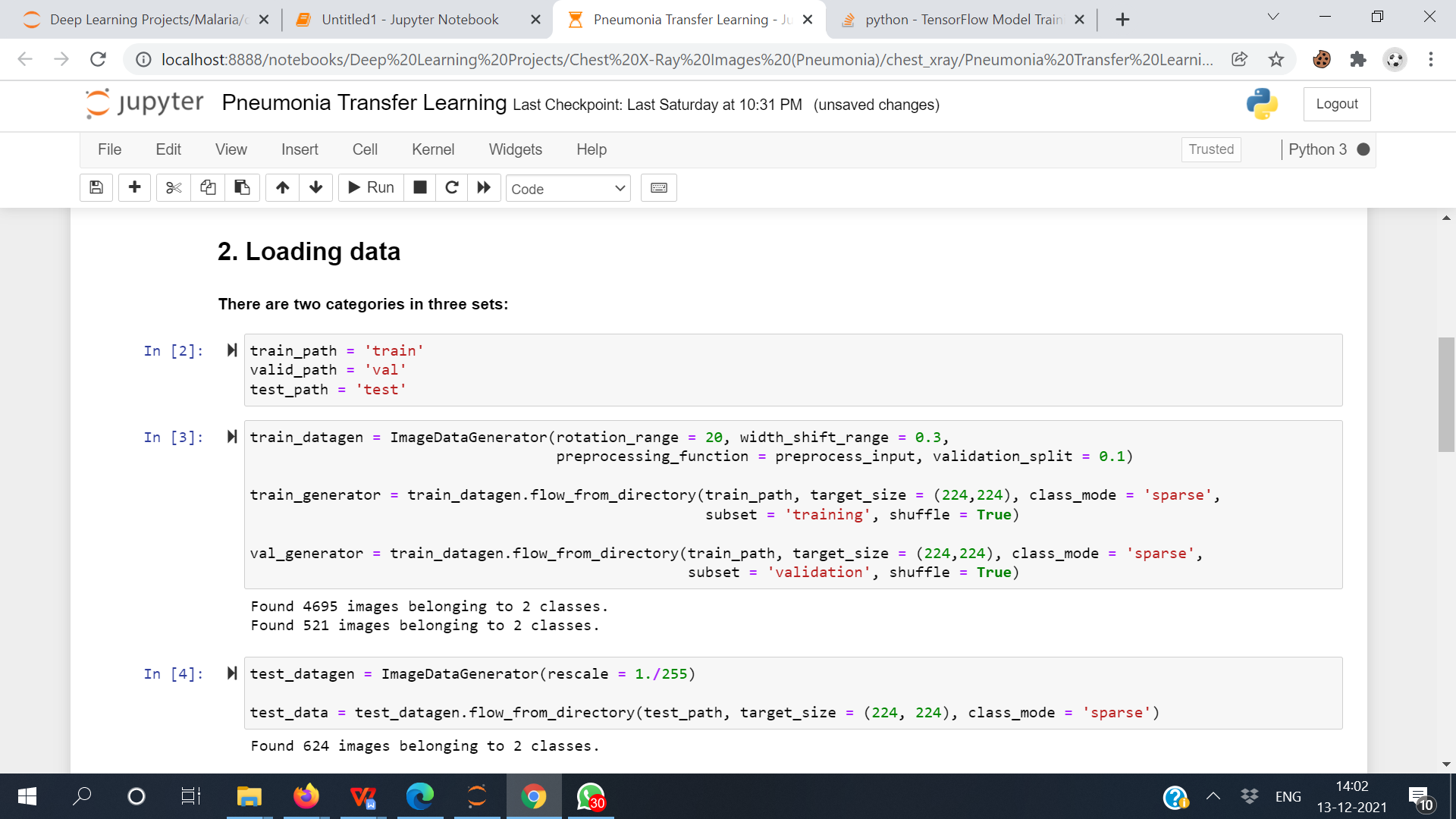


There are some other popular pre-trained models whose names you can find in the table below:-

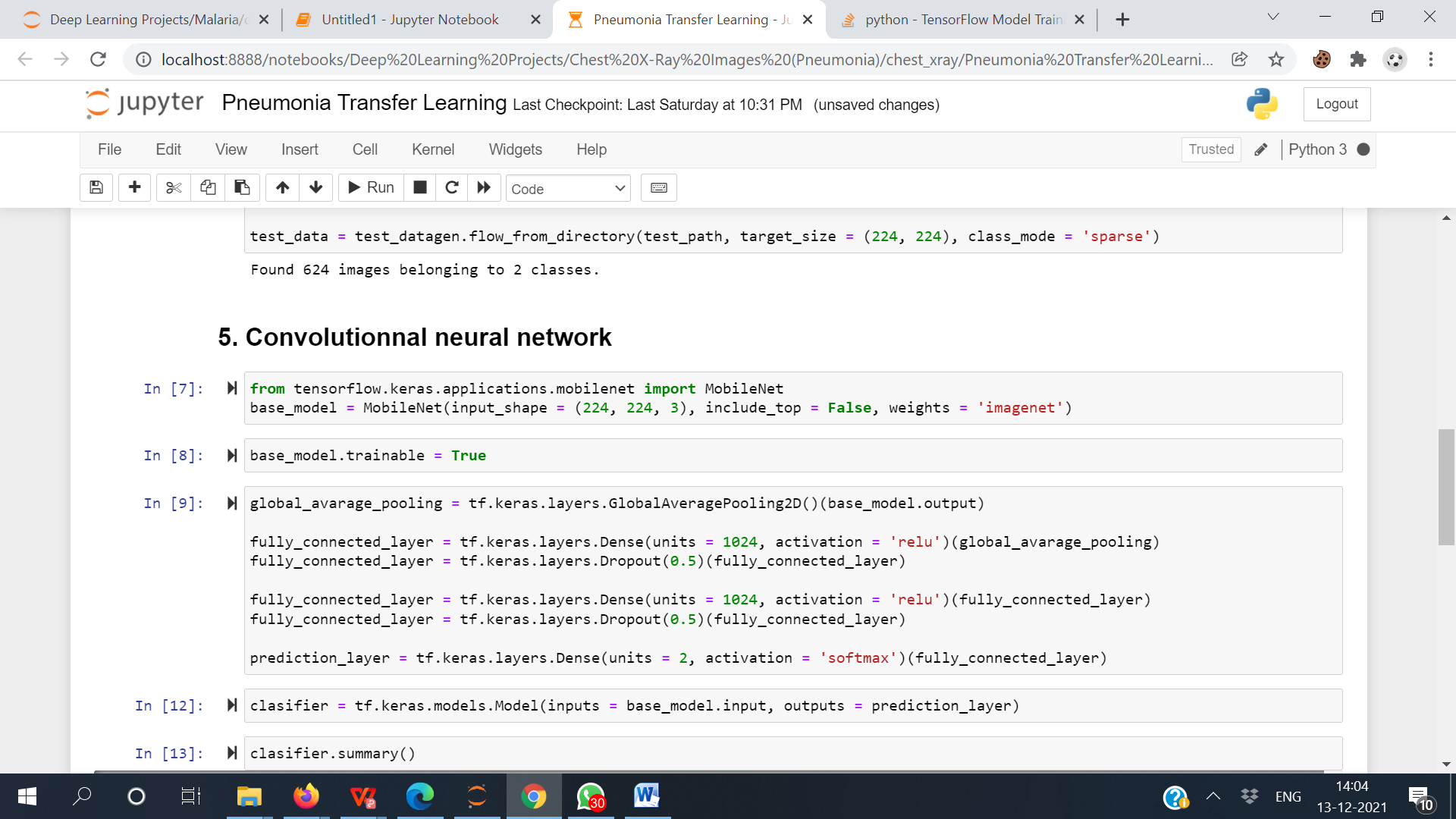
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Model** | **Size**  **(MB)** | **Top-1 Accuracy** | **Top-5**  **Accuracy** | **Parameters** | **Depth** | **Time (ms) per step (CPU)** | **Time (ms) per step (GPU)** |
| [Xception](https://keras.io/api/applications/xception) | 88 | 0.79 | 0.945 | 2,29,10,480 | 126 | 109.42 | 8.06 |
| [VGG16](https://keras.io/api/applications/vgg/" \l "vgg16-function) | 528 | 0.713 | 0.901 | 13,83,57,544 | 23 | 69.5 | 4.16 |
| [VGG19](https://keras.io/api/applications/vgg/" \l "vgg19-function) | 549 | 0.713 | 0.9 | 14,36,67,240 | 26 | 84.75 | 4.38 |
| [ResNet50](https://keras.io/api/applications/resnet/" \l "resnet50-function) | 98 | 0.749 | 0.921 | 2,56,36,712 | - | 58.2 | 4.55 |
| [ResNet101](https://keras.io/api/applications/resnet/" \l "resnet101-function) | 171 | 0.764 | 0.928 | 4,47,07,176 | - | 89.59 | 5.19 |
| [ResNet152](https://keras.io/api/applications/resnet/" \l "resnet152-function) | 232 | 0.766 | 0.931 | 6,04,19,944 | - | 127.43 | 6.54 |
| [ResNet50V2](https://keras.io/api/applications/resnet/" \l "resnet50v2-function) | 98 | 0.76 | 0.93 | 2,56,13,800 | - | 45.63 | 4.42 |
| [ResNet101V2](https://keras.io/api/applications/resnet/" \l "resnet101v2-function) | 171 | 0.772 | 0.938 | 4,46,75,560 | - | 72.73 | 5.43 |
| [ResNet152V2](https://keras.io/api/applications/resnet/" \l "resnet152v2-function) | 232 | 0.78 | 0.942 | 6,03,80,648 | - | 107.5 | 6.64 |
| [InceptionV3](https://keras.io/api/applications/inceptionv3) | 92 | 0.779 | 0.937 | 2,38,51,784 | 159 | 42.25 | 6.86 |
| [InceptionResNetV2](https://keras.io/api/applications/inceptionresnetv2) | 215 | 0.803 | 0.953 | 5,58,73,736 | 572 | 130.19 | 10.02 |
| [MobileNet](https://keras.io/api/applications/mobilenet) | 16 | 0.704 | 0.895 | 42,53,864 | 88 | 22.6 | 3.44 |
| [MobileNetV2](https://keras.io/api/applications/mobilenet/" \l "mobilenetv2-function) | 14 | 0.713 | 0.901 | 35,38,984 | 88 | 25.9 | 3.83 |
| [DenseNet121](https://keras.io/api/applications/densenet/" \l "densenet121-function) | 33 | 0.75 | 0.923 | 80,62,504 | 121 | 77.14 | 5.38 |
| [DenseNet169](https://keras.io/api/applications/densenet/" \l "densenet169-function) | 57 | 0.762 | 0.932 | 1,43,07,880 | 169 | 96.4 | 6.28 |
| [DenseNet201](https://keras.io/api/applications/densenet/" \l "densenet201-function) | 80 | 0.773 | 0.936 | 2,02,42,984 | 201 | 127.24 | 6.67 |
| [NASNetMobile](https://keras.io/api/applications/nasnet/" \l "nasnetmobile-function) | 23 | 0.744 | 0.919 | 53,26,716 | - | 27.04 | 6.7 |
| [NASNetLarge](https://keras.io/api/applications/nasnet/" \l "nasnetlarge-function) | 343 | 0.825 | 0.96 | 8,89,49,818 | - | 344.51 | 19.96 |
| [EfficientNetB0](https://keras.io/api/applications/efficientnet/" \l "efficientnetb0-function) | 29 | - | - | 53,30,571 | - | 46 | 4.91 |
| [EfficientNetB1](https://keras.io/api/applications/efficientnet/" \l "efficientnetb1-function) | 31 | - | - | 78,56,239 | - | 60.2 | 5.55 |
| [EfficientNetB2](https://keras.io/api/applications/efficientnet/" \l "efficientnetb2-function) | 36 | - | - | 91,77,569 | - | 80.79 | 6.5 |
| [EfficientNetB3](https://keras.io/api/applications/efficientnet/" \l "efficientnetb3-function) | 48 | - | - | 1,23,20,535 | - | 139.97 | 8.77 |
| [EfficientNetB4](https://keras.io/api/applications/efficientnet/" \l "efficientnetb4-function) | 75 | - | - | 1,94,66,823 | - | 308.33 | 15.12 |
| [EfficientNetB5](https://keras.io/api/applications/efficientnet/" \l "efficientnetb5-function) | 118 | - | - | 3,05,62,527 | - | 579.18 | 25.29 |
| [EfficientNetB6](https://keras.io/api/applications/efficientnet/" \l "efficientnetb6-function) | 166 | - | - | 4,32,65,143 | - | 958.12 | 40.45 |
| [EfficientNetB7](https://keras.io/api/applications/efficientnet/" \l "efficientnetb7-function) | 256 | - | - | 6,66,58,687 | - | 1578.9 | 61.62 |

**How to implement pre-trained models**

Lets use a simple pre-trained model to make you understand how to implement it. Here I take the reference of Kaggle Pneumonia dataset.



The above picture describes how you can read the images. Here the image dimension is set to (224,224,3). After some Image processing as required to get a good result, I have used a pre-trained model.



The above image shows how to import and use the pre-trained model in our problem statement. In the above picture, you can see that I have added two fully connected layers by myself to satisfy the purpose of the dataset. Lastly, you can see that I have merged the model and output layer into a single architecture with the name of a classifier.

You can use other pre-trained model to get a better accuracy.

**Notebook Link**

You can have a whole walkthrough the notebook via the links:-

<https://github.com/SubhankarSamanta295/Sophos/blob/main/Pneumonia%20Transfer%20Learning.ipynb>

<https://www.kaggle.com/subhankar007/pneumonia-transfer-learning>