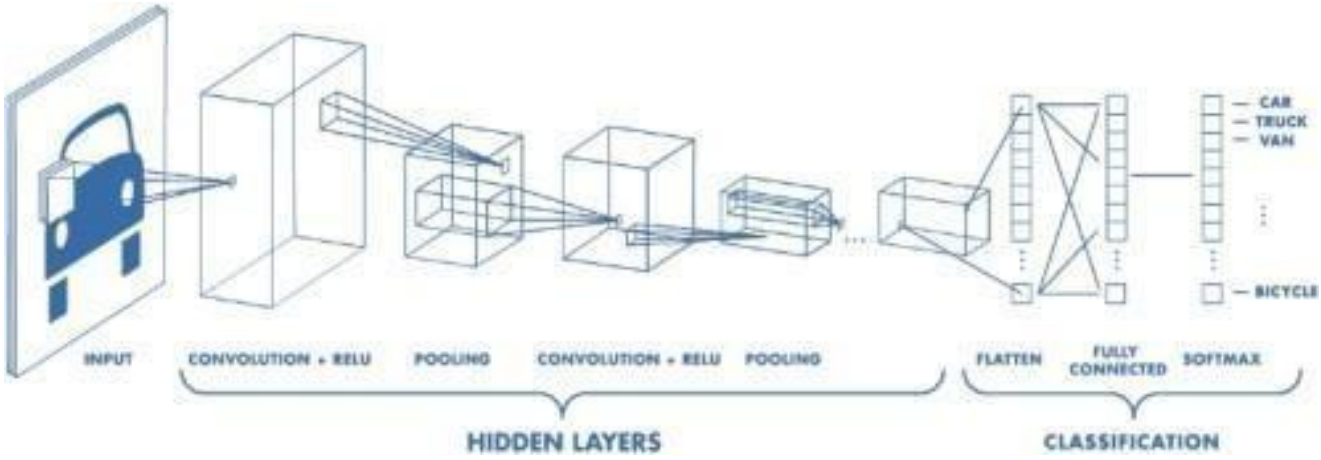
Prior Knowledge

Team ID	**PNT2022TMID52149**
Project Name	**Deep Learning Fundus Image Analysis for Early Detection of Diabetic Retinopathy**

Convolutional neural network:

Basics of the Classic CNN

How a classic CNN (Convolutional Neural Network) work?



Classic CNN architecture.

Convolutional neural networks. Sounds like a weird combination of biology and math with a little CS sprinkled in, but these networks have been some of the most influential innovations in the field of computer vision and image processing.

The Convolutional neural networks are regularized versions of multilayer perceptron (MLP). They were developed based on the working of the neurons of the animal visual cortex.

Visualization of an image by a computer

Binary image visualization.

Let's say we have a color image in JPG form and its size is 480 x 480. The representative array will be 480 x 480 x 3. Each of these numbers is given a value from 0 to 255 which describes the pixel intensity at that point. RGB intensity values of the image are visualized by the computer for processing.

The objective of using the CNN:

The idea is that you give the computer this array of numbers and it will output numbers that describe the probability of the image being a certain class (.80 for a cat, .15 for a dog, .05 for a bird, etc.). It

works similar to how our brain works. When we look at a picture of a dog, we can classify it as such if the picture has identifiable featuressuch as paws or 4 legs. In a similar way, the computer is able to perform image classification by looking for low-level features such as edges and curves and then building up to more abstract concepts through a series of convolutional layers. The computer uses low-level features obtained at the initial levels to generate high-level features such as paws or eyes to identify the object.

A Classic CNN:

Contents of a classic Convolutional Neural Network: -

1. Convolutional Layer.
2. Activation operation following each convolutional layer.

3. Pooling layer especially Max Pooling layer and also others based on the

requirement. 4.Finally Fully Connected Layer.

Convolution Operation

First Layer:

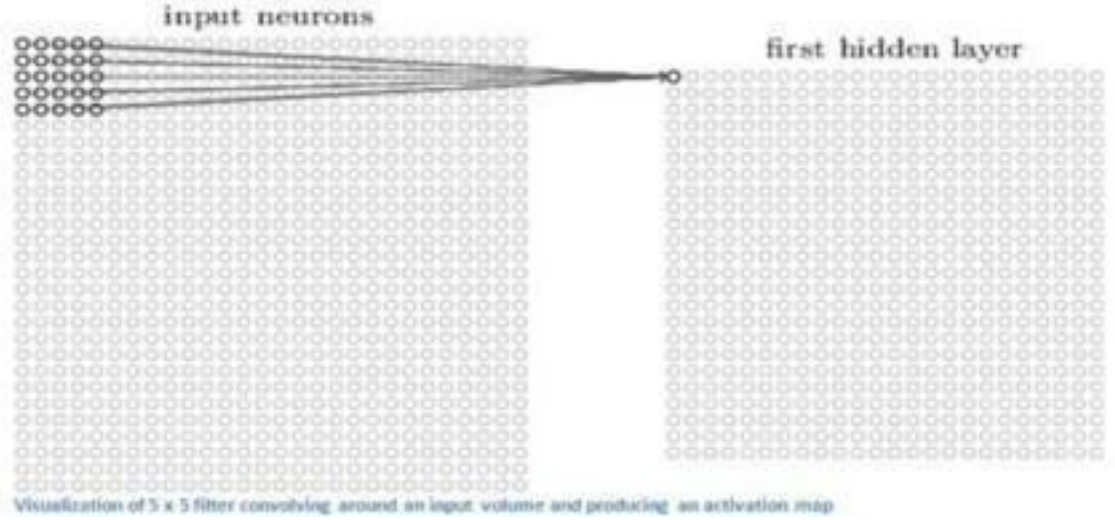
1. **Input to a convolutional layer**

The image is resized to an optimal size and is fed as input to the convolutional layer.

Let us consider the input as 32x32x3 array of pixel values

2. There exists a filter or neuron or kernel which lays over some of the pixels of the input image depending on the dimensions of the Kernel size.

Let the dimensions of the kernel of the filter be 5x5x3.

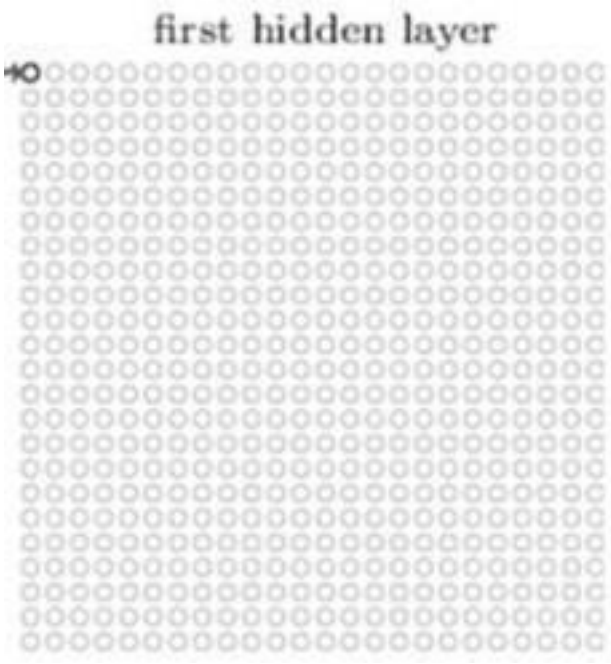


Visualization of 5 x 5 filter convolving around an input volume and producing an activation map

3. The Kernel actually slides over the input image, thus it is multiplying the values in the filter with the original pixel values of the image (aka computing **element-wise multiplications**).

The multiplications are summed up generating a single number for that particular receptive field and hence for sliding the kernel a total of 784 numbers are mapped to 28x28 array known as the **feature map**.

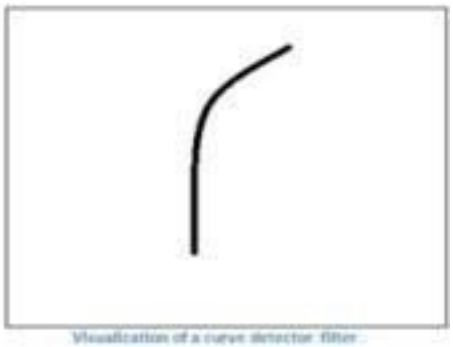
**Now if we consider two kernels of the same dimension then the obtained first layer feature map will be (28x28x2).



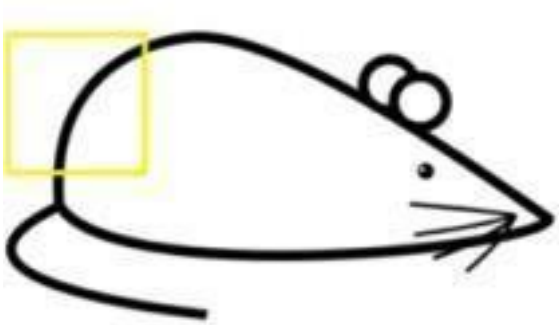
High-level Perspective

- Let us take a kernel of size (7x7x3) for understanding. Each of the kernels is considered to be a feature identifier, hence say that our filter will be a curve detector.

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

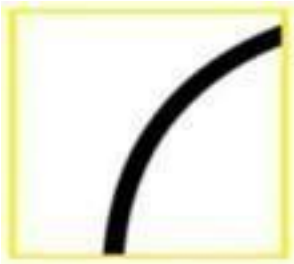


- **The original image and the visualization of the kernel on the image.**



Visualization of the filter on the image

Original image



Visualization of the receptive field

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

Pixel representation of the receptive field

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

The sum of the multiplication value that is generated is = 4*(50*30)+(20*30) = 6600 (large number)

- **Now when the kernel moves to the other part of the image.**

Visualization of the filter on the image

0	0	0	0	0	0	0
0	40	0	0	0	0	0
40	0	40	0	0	0	0
40	20	0	0	0	0	0
0	50	0	0	0	0	0
0	0	50	0	0	0	0
25	25	0	50	0	0	0

Pixel representation of receptive field

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

The sum of the multiplication value that is generated is = 0 (small number).

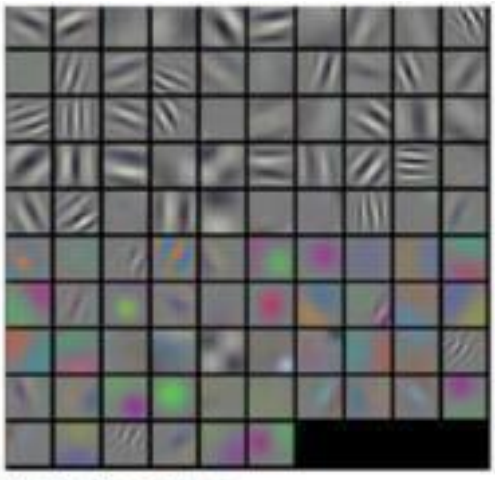
The use of the Small and the large value.

1. The value is much lower! This is because there wasn't anything in the image section that responded to the curve detector filter. Remember, the output of this convolution layer is an activation map. So, in the simple case of a one filter convolution (and if that filter is a curve detector), the activation map will show the areas in which there at most likely to be curved in the picture.

2. In the previous example, the top-left value of our 26 x 26 x 1 activation map (26 because of the 7x7 filter instead of 5x5) will be 6600. This high value means that it is likely that there is some sort of curve in the input volume that caused the filter to activate. The top right value in our activation map will be 0 because there wasn't anything in the input volume that caused the filter to activate. This is just for one filter.

3. This is just a filter that is going to detect lines that curve outward and to the right. We can have other filters for lines that curve to the left or for straight edges. The more filters, the greater the depth of the activation map, and the more information we have about the input volume.

In the picture, we can see some examples of actual visualizations of the filters of the first conv. layer of a trained network. Nonetheless, the main argument remains the same. The filters on the first layer convolve around the input image and "activate" (or compute high values) when the specific feature it is looking for is in the input volume.



Visualizations of filters

Sequential conv. layers after the first one.

1. When we go through another conv. layer, the output of the first conv. layer becomes the input of

the 2nd conv. layer.

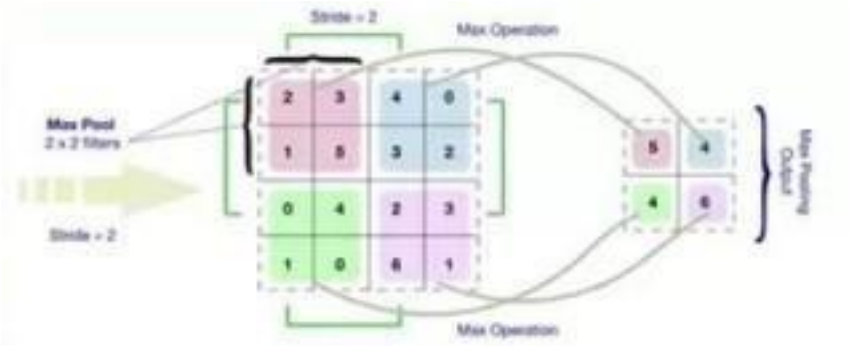
2. However, when we're talking about the 2nd conv. layer, the input is the activation map(s) that result from the first layer. So each layer of the input is basically describing the locations in the original image for where certain low-level features appear.

3. Now when you apply a set of filters on top of that (pass it through the 2nd conv. layer), the output will be activations that represent higher-level features. Types of these features could be semicircles (a combination of a curve and straight edge) or squares (a combination of several straight edges). As you go through the network and go through more conv. layers, you get activation maps that represent more and more complex features.

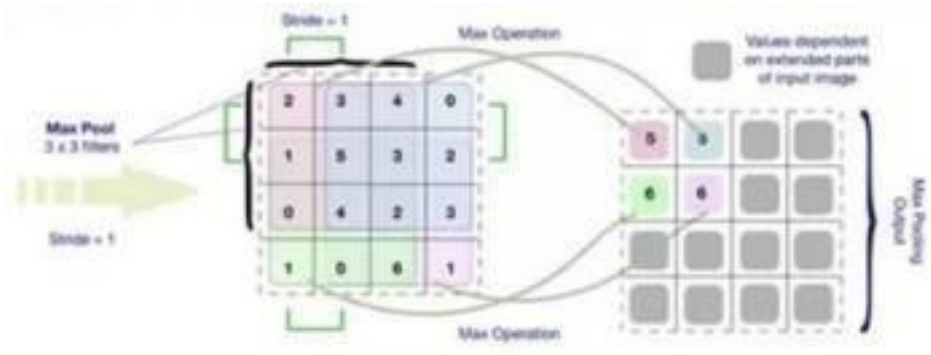
4. By the end of the network, you may have some filters that activate when there is handwriting in the image, filters that activate when they see pink objects, etc.

Pooling Operation.

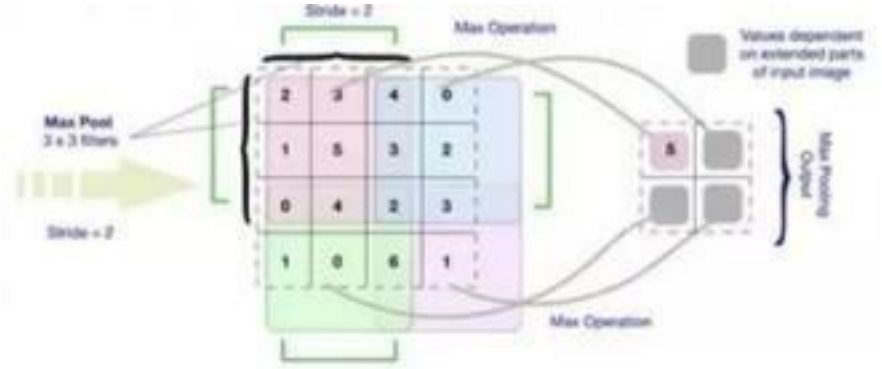
Max Pooling example.



2x2 filters with stride = 2 (maximum value) is considered



3x3 filters with stride = 1 (maximum value) is considered



3x3 filters with stride = 2 (maximum value) is considered

Fully Connected layer.

1. The way this fully connected layer works is that it looks at the output of the previous layer (which as we remember should represent the activation maps of high-level features) and the number of classes **N** (10 for digit classification).
2. For example, if the program is predicting that some image is a dog, it will have high values in the activation maps that represent high-level features like a paw or 4 legs, etc. Basically, an FC layer looks at what high level features most strongly correlate to a particular class and has particular weights so that when you compute the products between the weights and the previous layer, you get the correct probabilities for the different classes.

3. The output of a fully connected layer is as follows [0 .1 .1 .75 0 0 0 0 0 .05], then this represents a

10% probability that the image is a 1, a 10% probability that the image is a 2, a 75% probability that the image is a 3, and a 5% probability that the image is a 9 (Softmax approach) for digit classification.

Training.

§We know kernels also known as feature identifiers, used for identification of specific features. But how the kernels are initialized with the specific weights or how do the filters know what values to have.

Hence comes the important step of training. The training process is also known as backpropagation, which is further separated into 4 distinct sections or processes.

- Forward Pass
- Loss Function
- Backward Pass
- Weight Update

The Forward Pass:
For the first epoch or iteration of the training the initial kernels of the first conv. layer is initialized with random values. Thus after the first iteration output will be something like [.1.1.1.1.1.1.1.1.1], which does not give preference to any class as the kernels don't have specific weights.

The Loss Function:

The training involves images along with labels, hence the label for the digit **3** will be [0 0 0 1 0 0 0 0 0 0], whereas the output after a first epoch is very different, hence we will calculate loss (**MSE — Mean Squared Error**)

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

The objective is to minimize the loss, which is an optimization problem in calculus. It involves trying to adjust the weights to reduce the loss.

The Backward Pass:

It involves determining which weights contributed most to the loss and finding ways to adjust them so that the loss decreases. It is computed using dL/dW, where L is the loss and the W is the weights of the corresponding kernel.

The weight update:

This is where the weights of the kernel are updated using the following equation.

$$w = w_i - \eta \frac{dL}{dW}$$

w = Weight
w_i = Initial Weight
η = Learning Rate

Here the **Learning Rate** is chosen by the programmer. Larger value of the learning rate indicates

much larger steps towards optimization of steps and larger time to convolve to an optimized weight.

Testing.

Finally, to see whether or not our CNN works, we have a different set of images and labels (can't double dip between training and test!) and pass the images through the CNN. We compare the outputs to the ground truth and see if our network works!

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is an annual computer vision competition. Each year, teams compete on two tasks. The first is to detect objects within an image coming from *200* classes, which is called object localization. The second is to classify images, each labeled with one of *1000* categories, which is called image classification. VGG 16 was proposed by Karen Simonyan and Andrew Zisserman of the Visual Geometry Group Lab of Oxford University in 2014 in the paper "VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION". This model won 1 and 2 place in the above categories in the 2014 ILSVRC challenge.

This model achieves *92.7% top-5* test accuracy on the ImageNet dataset which contains *14* million images belonging to 1000 classes.
Objective: The ImageNet dataset contains images of fixed size of *224*224* and have RGB channels. So, we have a tensor of *(224, 224, 3)* as our input. This model process the input image and outputs for the corresponding class. Suppose we have a model that predicts that the image belongs to class 0 with probability *1*, *class 1* with probability *0.05*, *class 2* with probability *0.05*, class *3* with probability *0.03*, *class 780* with probability *0.72*, class *999* with probability *0.05* and all other class

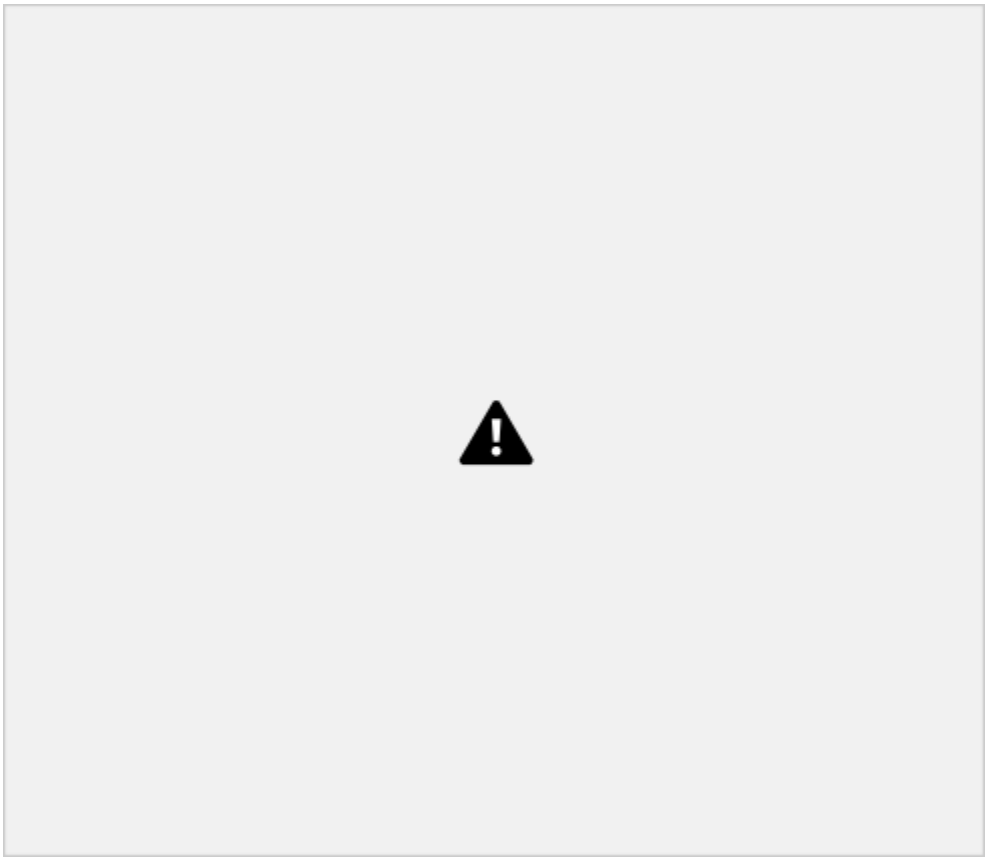
these probabilities add to *1*, we use softmax function.

G_{k}\, else \, d \, = \, 1 [/Tex]So, the loss function for this example is :

VGG Architecture: The input to the network is an image of dimensions *(224, 224, 3)*. The first two layers have *64* channels of a *3*3* filter size and the same padding. Then after a max pool layer of stride *(2, 2)*, two layers have convolution layers of 128 filter size and filter size *(3, 3)*. This is followed by a max-pooling layer of stride *(2, 2)* which is the same as the previous layer. Then there are *2* convolution layers of filter size *(3, 3)* and *256* filters. After that, there are *2* sets of *3* convolution layers and a max pool layer. Each has *512* filters of *(3, 3)* size with the same padding. This image is then passed to the

stack of two convolution layers. In these convolution and max pooling layers, the filters we use are of the size *3*3* instead of *11*11* in AlexNet and *7*7* in ZF-Net. In some of the layers, it also uses *1*1* pixel which is used to manipulate the number of input channels. There is a padding of *1-pixel* (same padding) done after each convolution layer to prevent the spatial feature of the image.

After the stack of convolution and max-pooling layer, we got a *(7, 7, 512)* feature map. We flatten this output to make it a *(1, 25088)* feature vector. After this there is *3 fully* connected layer, the first layer takes input from the last feature vector and outputs a *(1, 4096)* vector, the second layer also outputs a vector of size *(1, 4096)* but the third layer output a *1000* channels for *1000* classes of ILSVRC challenge i.e. 3rd fully connected layer is used to implement softmax function to classify 1000 classes. All the hidden layers use ReLU as its activation function. ReLU is more computationally efficient because it results in faster learning and it also decreases the likelihood of vanishing gradient problems.
Configuration: The table below listed different VGG architectures. We can see that there are 2 versions of VGG-16 (C and D). There is not much difference between them except for one that except for some convolution layers, *(3, 3)* filter size convolution is used instead of *(1, 1)*. These two contain *134* million and *138* million parameters respectively.



Object Localization In Image: To perform localization, we need to replace the class score by box location coordinates. A bounding box location is represented by the 4-D vector (center bounding

coordinates(x,y), height, width). There are two versions of localization architecture, one is bounding box is shared among different candidates (the output is *4* parameter vector) and the other is a bounding box is class-specific (the output is *4000* parameter vector). The paper experimented with

both approaches on VGG -16 (D) architecture. Here we also need to change loss from classification loss to regression loss functions (such as MSE) that penalize the deviation of predicted loss from the ground truth.

Results: VGG-16 was one of the best performing architectures in the ILSVRC challenge 2014.It was the runner up in the classification task with a top-5 classification error of *7.32%* (only behind GoogLeNet with a classification error of *6.66%*). It was also the winner of localization task with *25.32%* localization error.

Limitations Of VGG 16:

- It is very slow to train (the original VGG model was trained on Nvidia Titan GPU for 2-3 weeks).
- The size of VGG-16 trained imageNet weights is *528* MB. So, it takes quite a lot of disk space and bandwidth which makes it inefficient.
- 138 million parameters lead to exploding gradients problem.

Further advancements: Resnets are introduced to prevent exploding gradients problem that occurred in VGG-16.

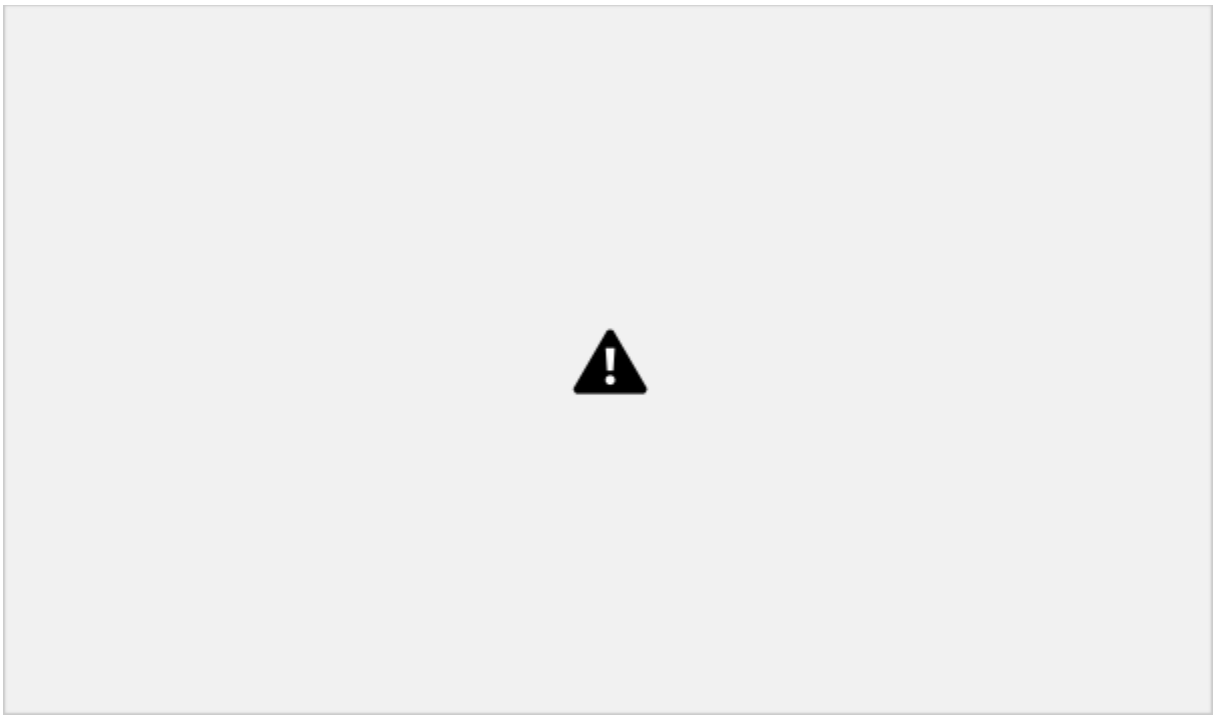
RESNET

Understanding and Coding a ResNet in Keras

Doing cool things with data!

ResNet, short for Residual Networks is a classic neural network used as a backbone for many computer vision tasks. This model was the winner of ImageNet challenge in 2015. The fundamental breakthrough with ResNet was it allowed us to train extremely deep neural networks with 150+layers successfully. Prior to ResNet training very deep neural networks was difficult due to the problem of vanishing gradients.

AlexNet, the winner of ImageNet 2012 and the model that apparently kick started the focus on deep learning had only 8 convolutional layers, the VGG network had 19 and Inception or GoogleNet had 22 layers and ResNet 152 had 152 layers. In this blog we will code a ResNet-50 that is a smaller version of ResNet 152 and frequently used as a starting point for transfer learning.



Revolution of Depth

However, increasing network depth does not work by simply stacking layers together. Deep networks are hard to train because of the notorious vanishing gradient problem — as the gradient is back-propagated to earlier layers, repeated multiplication may make the gradient extremely small. As a result, as the network goes deeper, its performance gets saturated or even starts degrading rapidly.

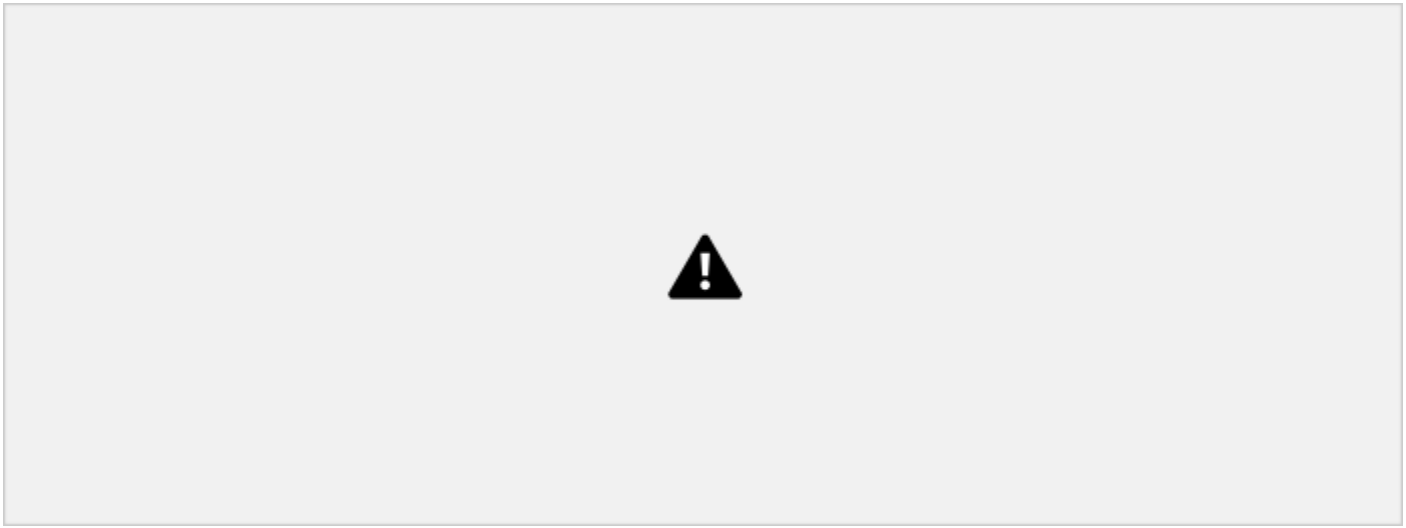
I learnt about coding ResNets from DeepLearning.AI course by Andrew Ng. I highly recommend this course.

On my Github repo, I have shared two notebooks one that codes ResNet from scratch as explained in DeepLearning.AI and the other that uses the pretrained model in Keras. I hope you pull the code and try it for yourself.

Skip Connection — The Strength of ResNet

ResNet first introduced the concept of skip connection. The diagram below illustrates skip connection. The figure on the left is stacking convolution layers together one after the other. On the right we still stack convolution layers as before but we now also add the original input to the output of the convolution

block. This is called skip connection



Skip Connection Image from DeepLearning.AI

It can be written as two lines of code :

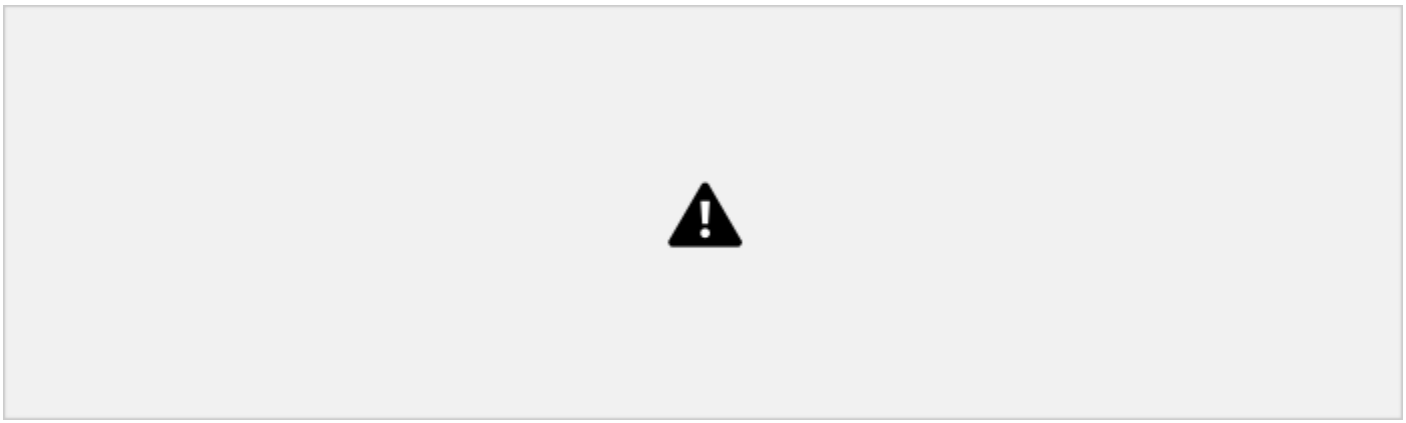
```
X_shortcut = X # Store the initial value of X in a variable
## Perform convolution + batch norm operations on XX = Add()([X, X_shortcut]) # SKIP Connection
```

The coding is quite simple but there is one important consideration — since X, X_shortcut above are two matrixes, you can add them only if they have the same shape. So if the convolution + batch norm operations are done in a way that the output shape is the same,then we can simply add them as shown

below.

When x and x_shortcut are the same shape

Otherwise, the x_shortcut goes through a convolution layer chosen such that the output from it is the same dimension as the output from the convolution block as shown below:



X_shortcut goes through convolution block

In the notebook on Github, the two functions identity_block and convolution_block implement above. These functions use Keras to implement Convolution and Batch Norm layers with ReLU activation. Skip connection is technically the one line `X = Add()([X, X_shortcut])`.

One important thing to note here is that the skip connection is applied before the RELU activation as shown in the diagram above. Research has found that this has the best results.

Why do Skip Connections work?

This is an interesting question. I think there are two reasons why Skip connections work here:

1. They mitigate the problem of vanishing gradient by allowing this alternate shortcut path for gradient to flow through

2. They allow the model to learn an identity function which ensures that the higher layer will perform at least as good as the lower layer, and not worse

Infact since ResNet skip connections are used in a lot more model architectures like the Fully Convolutional Network (FCN) andU-Net. They are used to flow information from earlier layers in the model to later layers. In these architectures they are used to pass information from the downsampling layers to the upsampling layers.

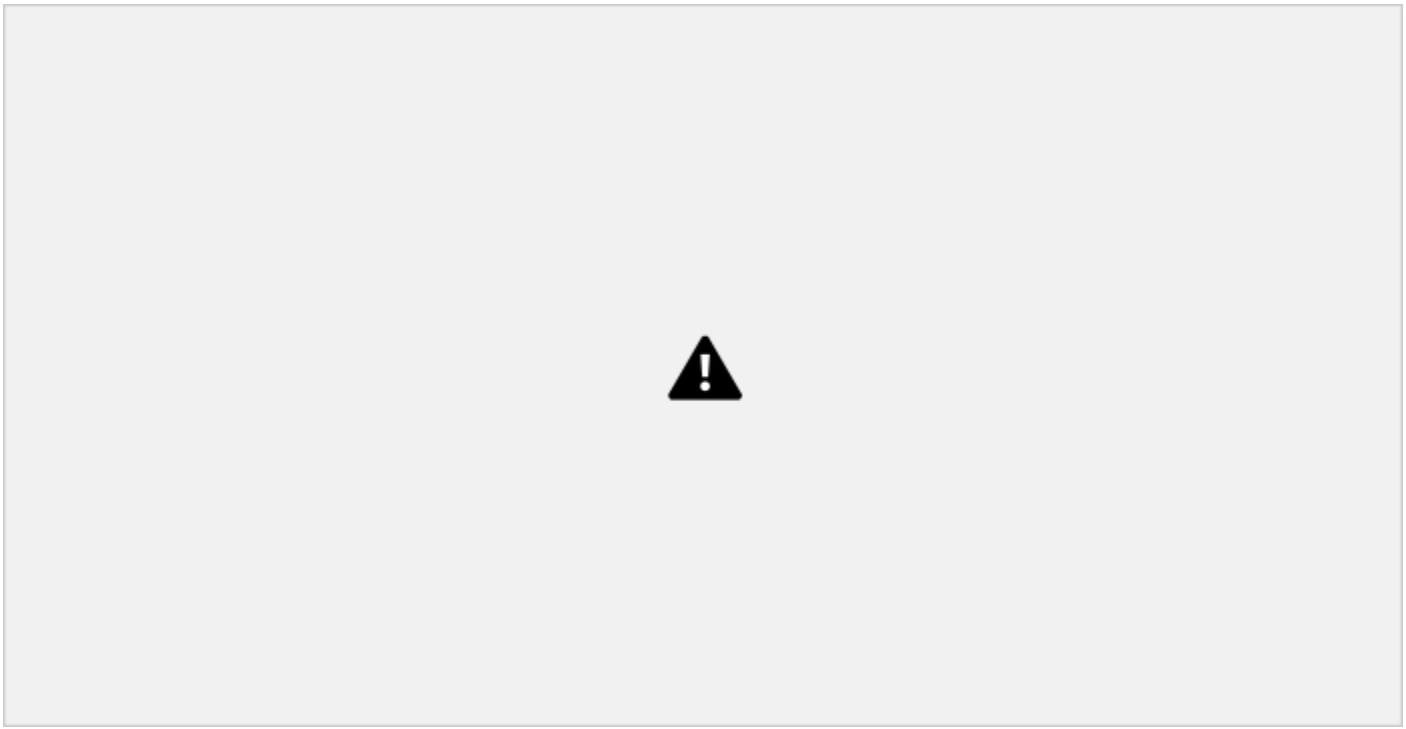
Testing the ResNet model we built

The identity and convolution blocks coded in the notebook are then combined to create a ResNet-50 model with the architecture shown below:

ResNet-50 Model

The ResNet-50 model consists of 5 stages each with a convolution and Identity block. Each convolution block has 3 convolution layers and each identity block also has 3 convolution layers. The ResNet-50 has over 23 million trainable parameters.

I have tested this model on the signs data set which is also included in my Github repo. This data set has hand images corresponding to 6 classes. We have 1080 train images and 120 test images.



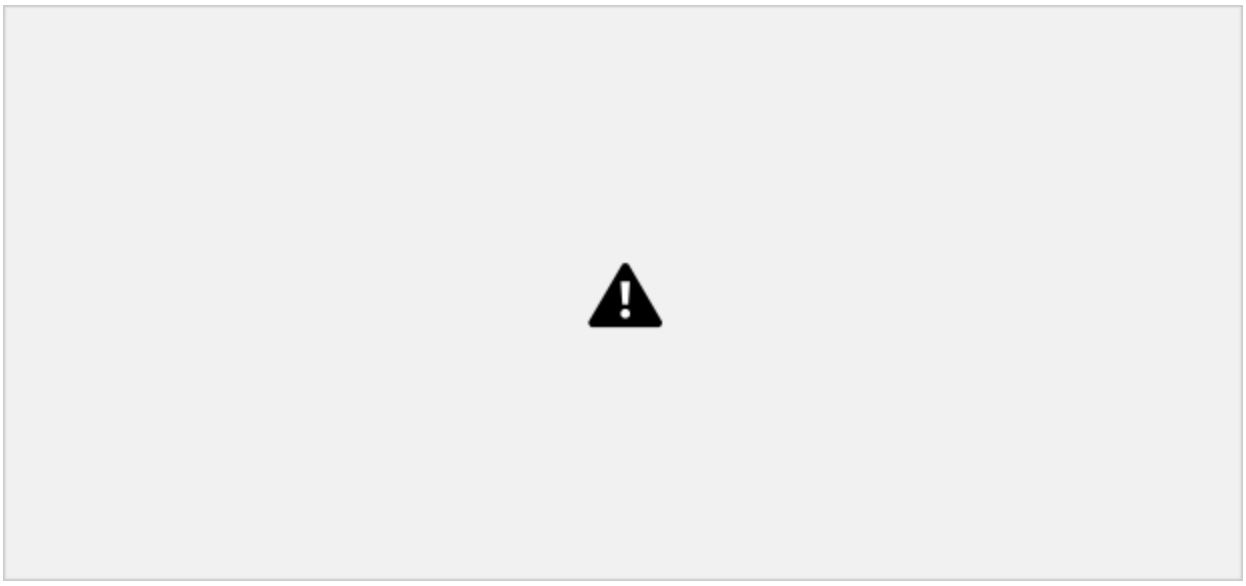
Signs Data Set

Our ResNet-50 gets to 86% test accuracy in 25 epochs of training. Not bad!

Building ResNet in Keras using pretrained library

I loved coding the ResNet model myself since it allowed me a better understanding of a network that I frequently use in many transfer learning tasks related to image classification, object localization, segmentation etc.

However for more regular use it is faster to use the pretrained ResNet-50 in Keras. Keras has many of these backbone models with their Imagenet weights available in its library.



Keras Pretrained Model

I have uploaded a notebook on my Github that uses Keras to load the pretrained ResNet-50. You can load the model with 1 line code:

```
base_model = applications.resnet50.ResNet50(weights= None, include_top=False, input_shape= (img_height,img_width,3))
```

Here weights=None since I want to initialize the model with random weights as I did on the ResNet-50 I coded. Otherwise I can also load the pretrained ImageNet weights. I set include_top=False to not include the final pooling and fully connected layer in the original model. I added Global Average Pooling and a

dense output layaer to the ResNet-50 model.

```
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dropout(0.7)(x)
predictions = Dense(num_classes, activation= 'softmax')(x)
model = Model(inputs = base_model.input, outputs = predictions)
```

As shown above Keras provides a very convenient interface to load the pretrained models but it is important to code the ResNet yourself as well at least once so you understand the concept

and can maybe apply this learning to another new architecture you are creating.

The Keras ResNet got to an accuracy of 75% after training on 100 epochs with Adam optimizer and a learning rate of 0.0001. The accuracy is a bit lower than our own coded model and I guess this has to do with weight initializations.

Keras also provides an easy interface for data augmentation so if you get a chance, try augmenting this data set and see if that results in better performance.

Conclusion

- ResNet is a powerful backbone model that is used very frequently in many computer vision tasks
- ResNet uses skip connection to add the output from an earlier layer to a later layer. This helps it mitigate the vanishing gradient problem
- You can use Keras to load their pretrained ResNet 50 or use the code I have shared to code ResNet yourself.

Introduction to Inception models

The Inception V3 is a deep learning model based on Convolutional Neural Networks, which is used for image classification. The inception V3 is a superior version of the basic model Inception V1 which was introduced as GoogLeNet in 2014. As the name suggests it was developed by a team at Google.

Inception V1

When multiple deep layers of convolutions were used in a model it resulted in the overfitting of the data. To avoid this from happening the inception V1 model uses the idea of using multiple filters of different sizes on the same level. Thus in the inception models instead of having deep layers, we have parallel layers thus making our model wider rather than making it deeper.

The Inception model is made up of multiple Inception modules.

The basic module of the Inception V1 model is made up of four parallel layers.

1. 1×1 convolution

2. 3×3 convolution
3. 5×5 convolution

4. 3×3 max pooling

Convolution - *The processof transforming an image by applying a kernel over each pixel and itslocal neighbors across the entire image.*
Pooling - *Pooling is the process used to reduce the dimensions of the feature map. There are different types of pooling but the most common ones are max pooling and average pooling.* Here, different sizes of convolutions are performed to capture different sizes of information in the Picture.

Naive Form

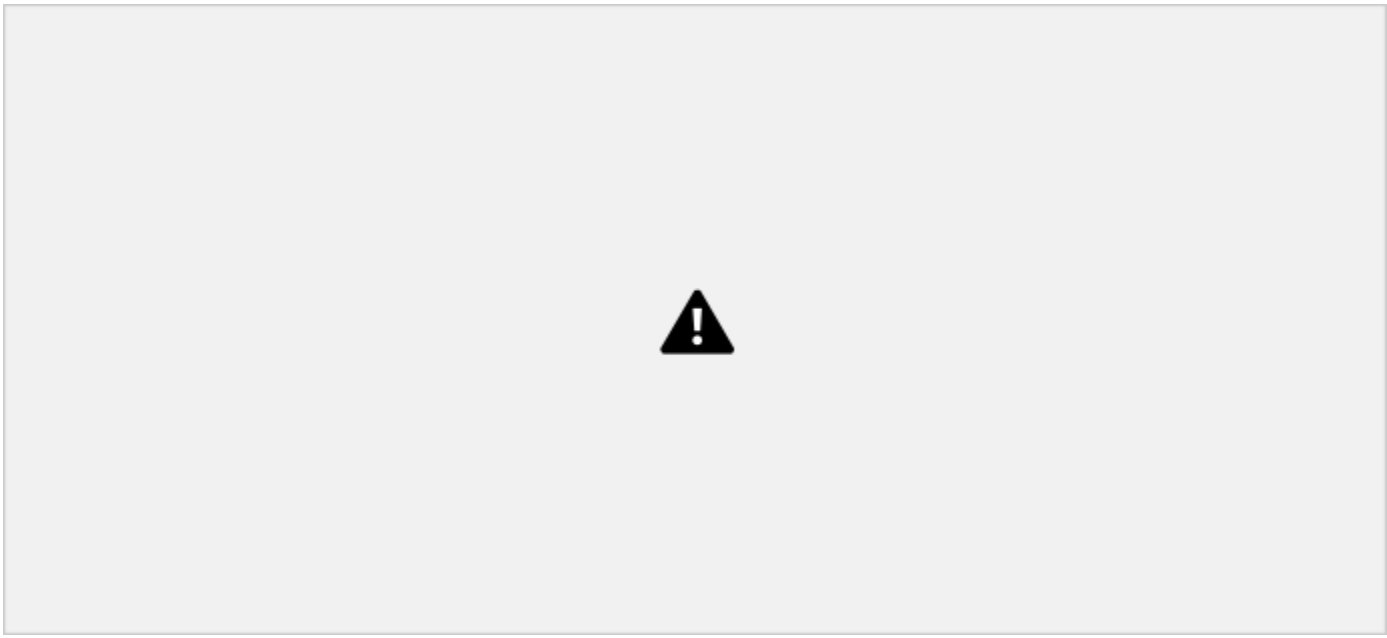
This module of the Inception V1 is called the Naive form. One of the drawbacks of this naive form is that even the 5×5 convolutional layer is computationally pretty expensive i.e. time-consuming and requires high computational power.

To overcome this the authors added a 1×1 convolutional layer before each convolutional layer, which results in reduced dimensions of the network and faster computations.

After adding the dimension reductionsthe module lookslike this.

This is the building block of the Inception V1 model Architecture. The Inception V1 architecture

model was better than most other models at that time. We can see that it has a very minimum error percentage.



Comparison of Inception V1 with other models.

What makes the Inception V3 model better?

The inception V3 is just the advanced and optimized version of the inception V1 model. The Inception V3 model used several techniques for optimizing the network for better model adaptation.

- It has higher efficiency
- It has a deeper network compared to the Inception V1 and V2 models, but its speed isn't compromised. • It is computationally less expensive.
- It uses auxiliary Classifiers as regularizes.

Inception V3 Model Architecture

The inception v3 model was released in the year 2015, it has a total of 42 layers and a lower error rate than its predecessors. Let's look at what are the different optimizations that make the inception V3 model better.

The major modifications done on the Inception V3 model are

1. Factorization into Smaller Convolutions
2. Spatial Factorization into Asymmetric Convolutions

3. Utility of Auxiliary Classifiers

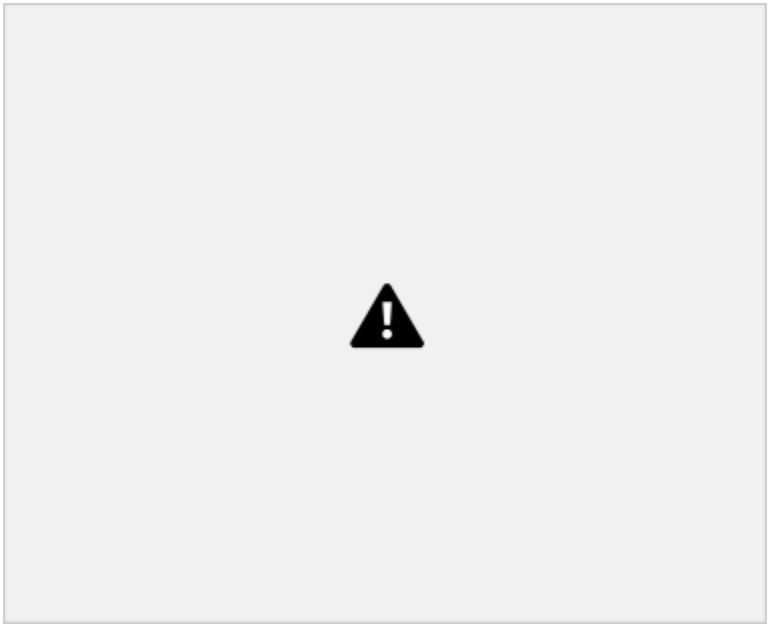
4. Efficient Grid Size Reduction

Let's how each one of these optimizations was implemented and how it improved the model.

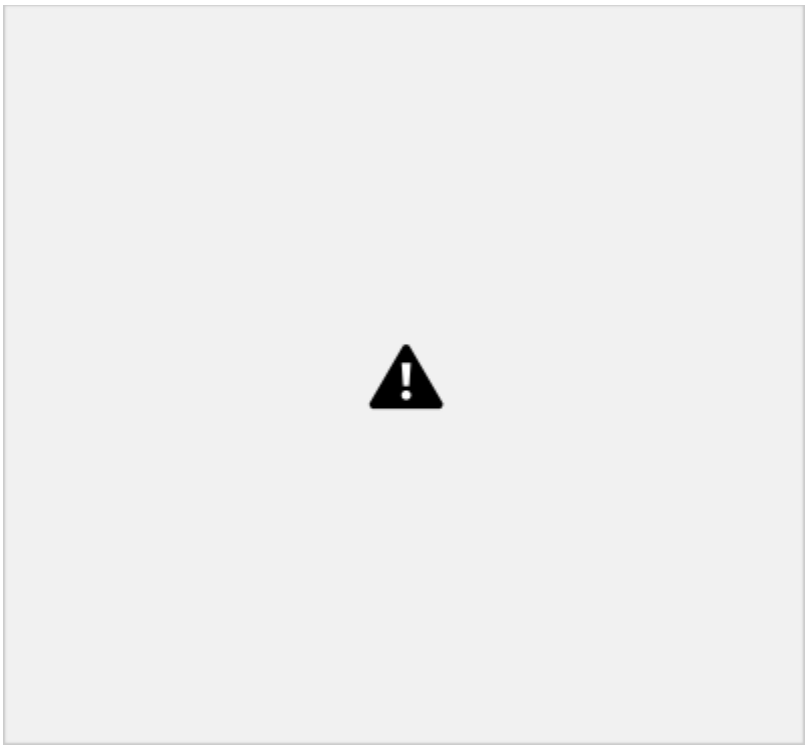
Factorization into Smaller Convolutions

One of the major assets of the Inception V1 model was the generous dimension reduction. To make it even better, the larger Convolutions in the model were factorized into smaller Convolutions.

For example, consider the basic module of the inception V1 module.

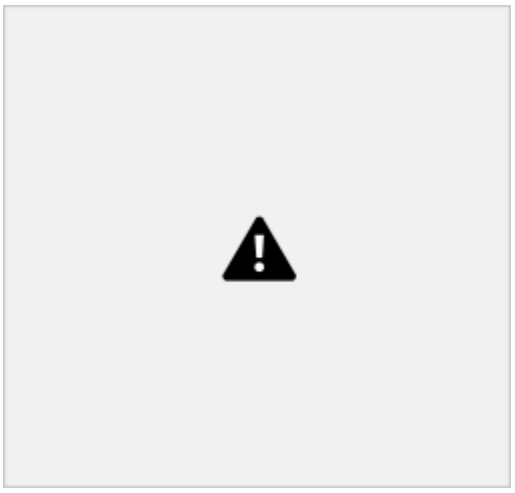


It has a 5×5 convolutional layer which was computationally expensive as said before. So to reduce the computational cost the 5×5 convolutional layer was replaced by two 3×3 convolutional layers as shown below.



Module 2

To understand it better see how the process of using two 3×3 convolutions reduces the number of parameters.

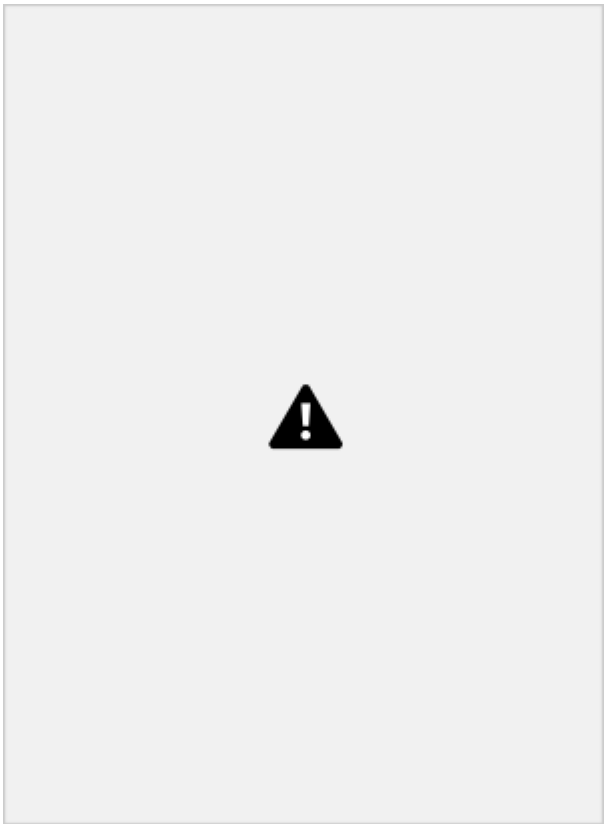


As a result of the reduced number of parameters the computational costs also reduce. This factorization of larger convolutions into smaller convolutions resulted in a relative gain of 28%.

Spatial Factorization into Asymmetric Convolutions

Even though the larger convolutions are factorized into smaller convolutions. You may wonder what if we can factorize furthermore for example to a 2×2 convolution. But, a better alternative to make the model more efficient was Asymmetric convolutions.

Asymmetric convolutions are of the form n×1.
So, what they did is replace the 3×3 convolutions with a 1×3 convolution followed by a 3×1 convolution. Doing so is the same as sliding a two-layer network with the same receptive field as in a 3×3 convolution.

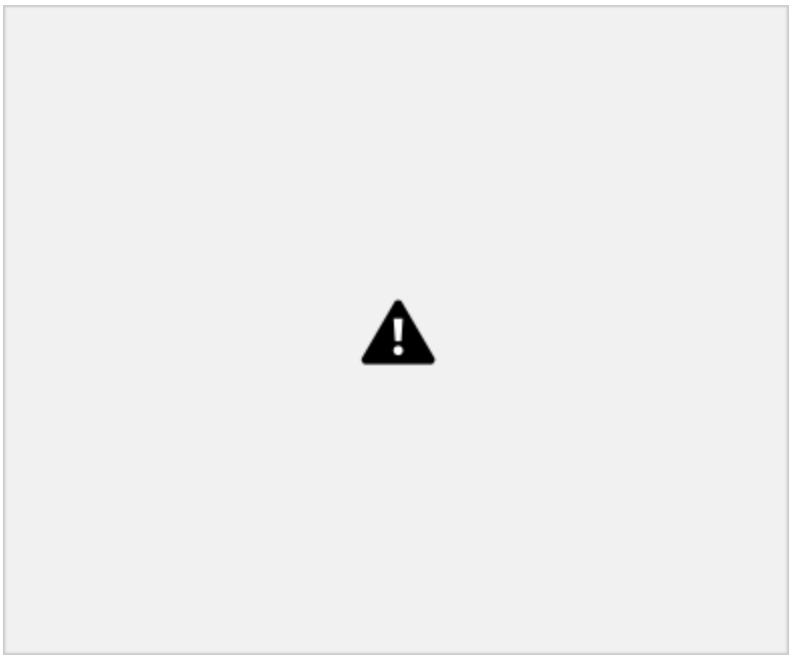


Module 2

Structure of Asymmetric Convolutions

The two-layer solution is 33% cheaper for the same number of output filters if the number of input and output filters is equal.

After applying the first two optimization techniques the inception module looks like this.



Module 3

Utility of Auxiliary classifiers

The objective of using an Auxiliary classifier is to improve the convergence of very deep neural networks. The auxiliary classifier is mainly used to combat the vanishing gradient problem in very deep networks.

The auxiliary classifiers didn't result in any improvement in the early stages of the training. But towards the end, the network with auxiliary classifiers showed higher accuracy compared to the network without auxiliary classifiers.

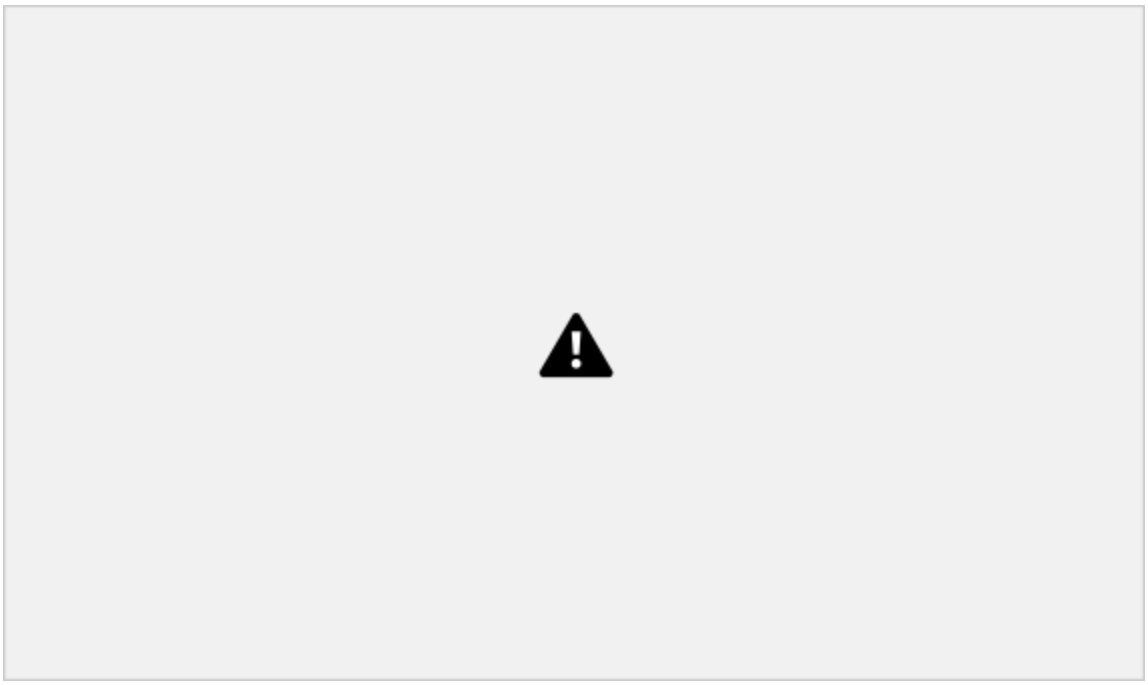
Thus the auxiliary classifiers act as a regularizer in Inception V3 model architecture.

Efficient Grid Size Reduction

Traditionally max pooling and average pooling were used to reduce the grid size of the feature maps. In the inception V3 model, in order to reduce the grid size efficiently the activation dimension of the network filters is expanded.

For example, if we have a d×d grid with k filters after reduction it results in a $d/2 \times d/2$ grid with 2k filters.

And this is done using two parallel blocks of convolution and pooling later concatenated.



The above image shows how the grid size is reduced efficiently while expanding the filter banks.

The final Inception V3 model

After performing all the optimizations the final Inception V3 model looks like this
In total, the inception V3 model is made up of 42 layers which is a bit higher than the previous inception V1 and V2 models. But the efficiency of this model is really impressive. We will get to it in a bit, but before it let's just see in detail what are the components the Inception V3 model is made of.

TYPE	PATCH / STRIDE SIZE	INPUT SIZE
Conv	3×3/2	299×299×3
Conv	3×3/1	149×149×32
Conv padded	3×3/1	147×147×32
Pool	3×3/2	147×147×64
Conv	3×3/1	73×73×64
Conv	3×3/2	71×71×80

Conv	3×3/1	35×35×192
3 × Inception	Module 1	35×35×288
5 × Inception	Module 2	17×17×768
2 × Inception	Module 3	8×8×1280
Pool	8 × 8	8 × 8 × 2048
Linear	Logits	1 × 1 × 2048
Softmax	Classifier	1 × 1 × 1000

The above table describes the outline of the inception V3 model. Here, the output size of each module is the input size of the next module.

Performance of Inception V3

As expected the inception V3 had better accuracy and less computational cost compared to the previous Inception version.

Xception

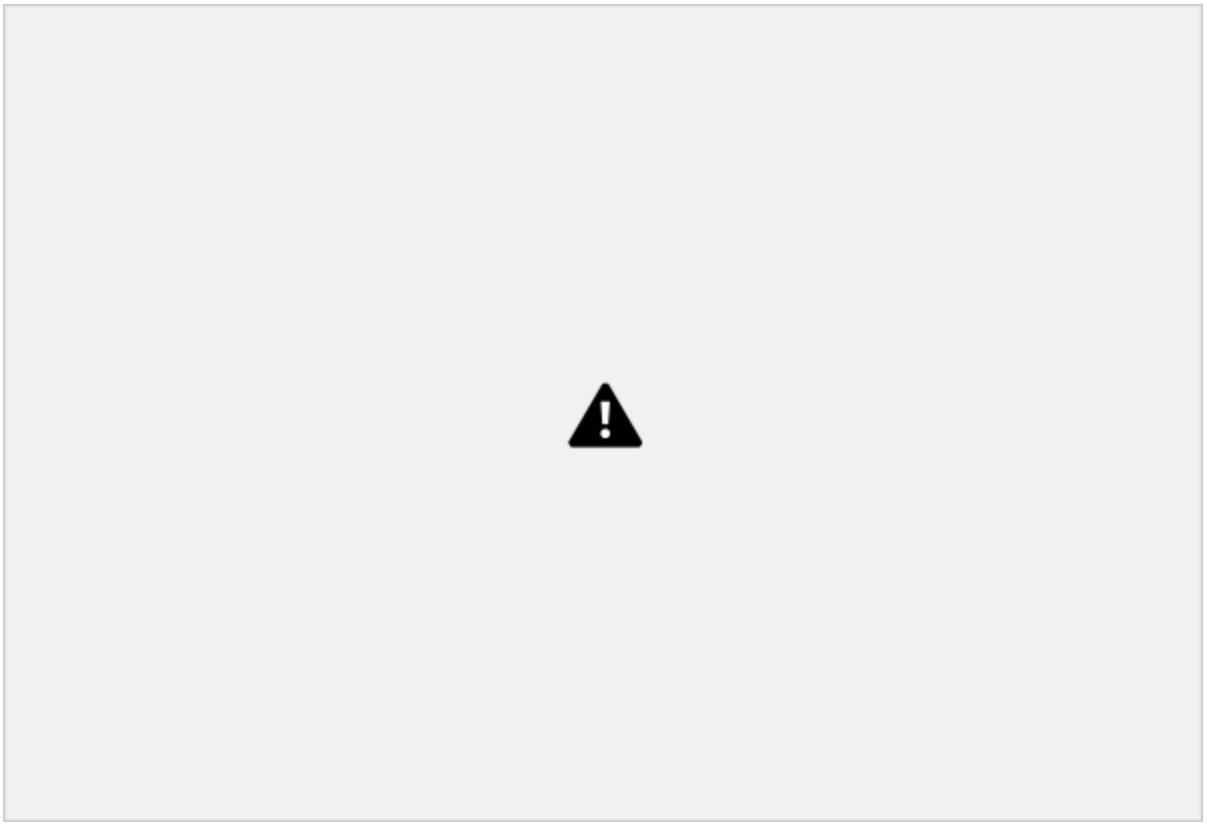


Figure 6: The Xception architecture.

Xception was proposed by none other than[François Chollet](#) himself, the creator and chief maintainer of the Keras library.

Xception is an extension of the Inception architecture which replaces the standard Inception modules with depthwise separable convolutions.
The original publication, *Xception: Deep Learning with Depthwise Separable Convolutions* can be found [here](#).

Xception sports the smallest weight serialization at only 91MB.

What about SqueezeNet?

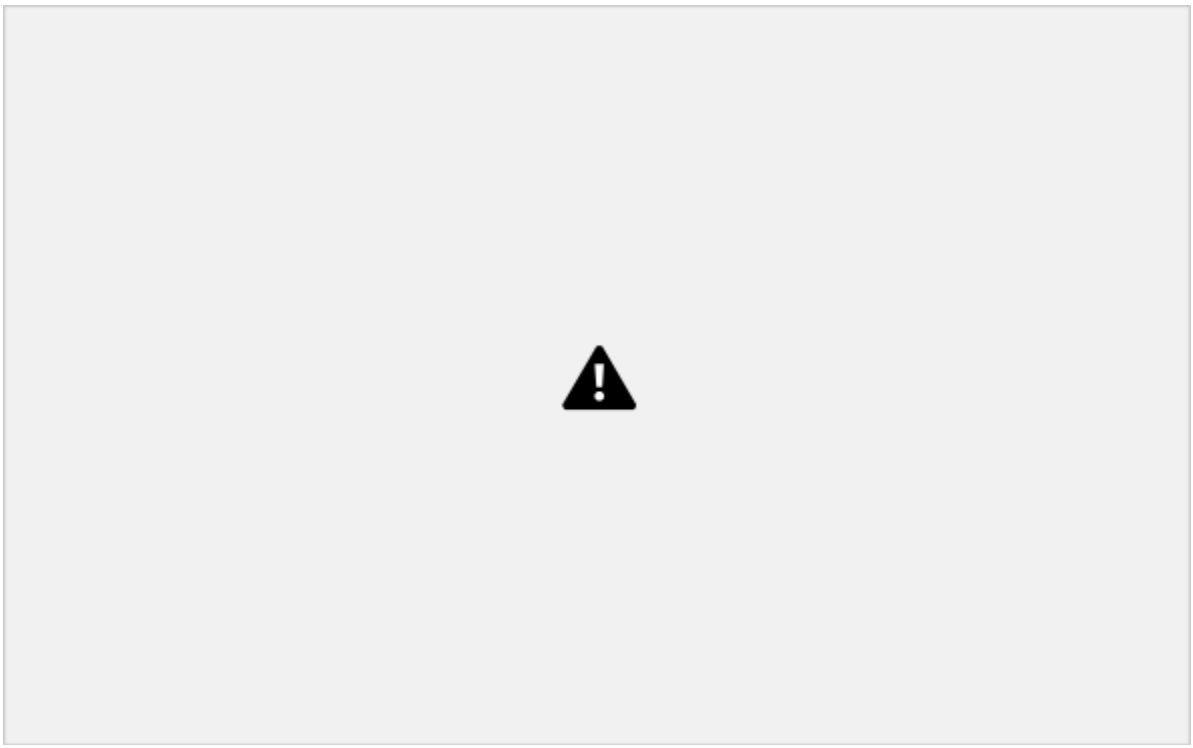


Figure 7: The “fire” module in SqueezeNet, consisting of a “squeeze” and an “expand”. (Iandola et al., 2016).

For what it’s worth, the SqueezeNet architecture can obtain AlexNet-level accuracy (~57% rank-1 and ~80% rank-5) at only 4.9MB through the usage of “fire” modules that “squeeze” and “expand”.

While leaving a small footprint, SqueezeNet can also be *very* tricky to train.

That said, I demonstrate how to train SqueezeNet from scratch on the ImageNet dataset inside my upcoming book, ***Deep Learning for Computer Vision with Python***.

Configuring your development environment

To configure your system for this tutorial, I recommend following either of these tutorials:

- ***How to install TensorFlow 2.0 on Ubuntu***
- ***How to install TensorFlow 2.0 on macOS***

Either tutorial will help you configure you system with all the necessary software for this blog post in a convenient Python virtual environment.

Please note that PyImageSearch does not recommend or support Windows for CV/DL projects.

Classifying images with VGGNet, ResNet, Inception, and Xception with Python and Keras

Let's learn how to classify images with pre-trained Convolutional Neural Networks using the Keras library.
Open up a new file, name it

classify_image.py

, and insert the following code:

VGGNet, ResNet, Inception, and Xception with Keras

```
# import the necessary packages
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.applications import InceptionV3
from tensorflow.keras.applications import Xception # TensorFlow ONLY
from tensorflow.keras.applications import VGG16
from tensorflow.keras.applications import VGG19
from tensorflow.keras.applicationsimport imagenet_utils
from tensorflow.keras.applications.inception_v3 import preprocess_input
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.preprocessing.image import load_img
import numpy as np
import argparse
import cv2
```

Lines 2-13 import our required Python packages. As you can see, most of the packages are part of the Keras library.

Specifically, **Lines 2-6** handle importing the Keras implementations of ResNet50, Inception V3,

Xception, VGG16, and VGG19, respectively.

Please note that the Xception network is compatible *only with the TensorFlow backend* (the class will throw an error if you try to instantiate it with a Theano backend).
Line 7 gives us access to the

imagenet_utils

sub-module, a handy set of convenience functions that will make pre-processing our input images and decoding output classifications easier.

The remainder of the imports are other helper functions, followed by NumPy for numerical processing and

cv2

for our OpenCV bindings.

Next, let's parse our command line arguments:

VGGNet, ResNet, Inception, and Xception with Keras

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required=True,
help="path to the input image")
ap.add_argument("-model", "--model", type=str, default="vgg16",
help="name of pre-trained network to use")
args = vars(ap.parse_args())
```

We'll require only a single command line argument,

--image

, which is the path to our input image that we wish to classify.

We'll also accept an optional command line argument,

--model

, a string that specifies which pre-trained Convolutional Neural Network we would like to use — this value defaults to

vgg16

for the VGG16 network architecture.

Given that we accept the name of our pre-trained network via a command line argument, we need to define a Python dictionary that maps the model names (strings) to their actual Keras classes:

→ Launch Jupyter Notebook on Google Colab → Launch Jupyter Notebook on Google Colab

VGGNet, ResNet, Inception, and Xception with Keras

```
# define a dictionary that maps model names to their classes
# inside Keras
MODELS = {
"vgg16": VGG16,
"vgg19": VGG19,
"inception": InceptionV3,
"xception": Xception, # TensorFlow ONLY
"resnet": ResNet50
}
# esnure a valid model name was supplied via command line argument
if args["model"] not in MODELS.keys():
raise AssertionError("The --model command line argument should "
"be a key in the `MODELS` dictionary")
```

Lines 25-31 defines our

MODELS

dictionary which maps a model name string to the corresponding class.

If the

`--model`

name is not found inside

`MODELS`

, we'll raise an

`AssertionError`

(**Lines 34-36**).

A Convolutional Neural Network takes an image as an input and then returns a set of probabilities corresponding to the class labels as output.

Typical input image sizes to a Convolutional Neural Network trained on ImageNet are *224×224*, *227×227*, *256×256*, and *299×299*; however, you may see other dimensions as well.

VGG16, VGG19, and ResNet all accept *224×224* input images while Inception V3 and Xception require *299×299* pixel inputs, as demonstrated by the following code block:

`VGGNet, ResNet, Inception, and Xception with Keras`

```
# initialize the input image shape (224x224 pixels) along with
# the pre-processing function (this might need to be changed
# based on which model we use to classify our image)
inputShape = (224, 224)
preprocess = imagenet_utils.preprocess_input
# if we are using the InceptionV3 or Xception networks, then we
# need to set the input shape to (299x299) [rather than (224x224)]
# and use a different image pre-processing function
if args["model"] in ("inception", "xception"):
inputShape = (299, 299)
```

preprocess = preprocess_input

Here we initialize our

inputShape

to be *224×224* pixels. We also initialize our

preprocess

function to be the standard

preprocess_input

from Keras (which performs mean subtraction).
However, if we are using Inception or Xception, we need to set the

inputShape

to *299×299* pixels, followed by updating

preprocess

to use a ***separate pre-processing function*** that **performs a different type of scaling**.

The next step is to load our pre-trained network architecture weights from disk and instantiate our model:

→ Launch Jupyter Notebook on Google Colab → Launch Jupyter Notebook on Google Colab

VGGNet, ResNet, Inception, and Xception with Keras

```
# load our the network weights from disk (NOTE: if this is the #
first time you are running this script for a given network, the
# weights will need to be downloaded first -- depending on which
# network you are using, the weights can be 90-575MB, so be
# patient; the weights will be cached and subsequent runs of this
# script will be *much* faster)
print("[INFO] loading {}...".format(args["model"]))
```

```
Network = MODELS[args["model"]]
model = Network(weights="imagenet")
```

Line 58 uses the

`MODELS`

dictionary along with the

`--model`

command line argument to grab the correct

`Network`

class.

The Convolutional Neural Network is then instantiated on **Line 59** using the pre-trained ImageNet weights;

Note: *Weights for VGG16 and VGG19 are > 500MB. ResNet weights are ~100MB, while Inception and Xception weights are between 90-100MB. If this is the* ***first*** *time you are running this script for a given network, these weights will be (automatically) downloaded and cached to your local disk. Depending on your internet speed, this may take awhile. However, once the weights are downloaded, they will* ***not*** *need to be downloaded again, allowing subsequent runs of*

`classify_image.py`

to be ***much*** *faster.*

Our network is now loaded and ready to classify an image — we just need to prepare this image for classification:

`VGGNet, ResNet, Inception, and Xception with Keras`

```
# load the input image using the Keras helper utility while ensuring #
the image is resized to `inputShape`, the required input dimensions

# for the ImageNet pre-trained network

print("[INFO] loading and pre-processing image...")
```

```
image = load_img(args["image"], target_size=inputShape)
image = img_to_array(image)
# our input image is now represented as a NumPy array of shape
# (inputShape[0], inputShape[1], 3) however we need to expand the #
dimension by making the shape (1, inputShape[0], inputShape[1], 3) #
so we can pass it through the network
image = np.expand_dims(image, axis=0)
# pre-process the image using the appropriate function based on the
# model that has been loaded (i.e., mean subtraction, scaling, etc.)
image = preprocess(image)
```

Line 65 loads our input image from disk using the supplied

`inputShape`

to resize the width and height of the image.

Line 66 converts the image from a PIL/Pillow instance to a NumPy array.

Our input image is now represented as a NumPy array with the shape

`(inputShape[0], inputShape[1], 3)`

.

However, we typically train/classify images in *batches*with Convolutional Neural Networks, so we need to add an extra dimension to the array via

`np.expand_dims`

on **Line 72**.

After calling
`np.expand_dims`

the

image

has the shape

(1, inputShape[0], inputShape[1], 3)

. Forgetting to add this extra dimension will result in an error when you

call .predict

of the

model

.

Lastly, **Line 76** calls the appropriate pre-processing function to perform mean

subtraction/scaling. We are now ready to pass our image through the network and obtain the

output classifications: → Launch Jupyter Notebook on Google Colab → Launch Jupyter

Notebook on Google Colab

VGGNet, ResNet, Inception, and Xception with Keras

```
# classify the image
print("[INFO] classifying image with '{}'...".format(args["model"]))
preds = model.predict(image)
P = imagenet_utils.decode_predictions(preds)
# loop over the predictions and display the rank-5 predictions +
# probabilities to our terminal
for (i, (imagenetID, label, prob)) in enumerate(P[0]):
print("{}. {}: {:.2f}%".format(i + 1, label, prob * 100))
```

A call to

.predict

on **Line 80** returns the predictions from the Convolutional Neural Network.
Given these predictions, we pass them into the ImageNet utility function

.decode_predictions

to give us a list of ImageNet class label IDs, “human-readable” labels, and the probability associated with the labels.

The top-5 predictions (i.e., the labels with the largest probabilities) are then printed to our terminal on **Lines 85 and 86**.

The last thing we’ll do here before we close out our example is load our input image from disk via OpenCV, draw the #1 prediction on the image, and finally display the image to our screen:

→ Launch Jupyter Notebook on Google Colab → Launch Jupyter Notebook on Google Colab

VGGNet, ResNet, Inception, and Xception with Keras

```
# load the image via OpenCV, draw the top prediction on the image,
# and display the image to our screen
orig = cv2.imread(args["image"])
(imagenetID, label, prob) = P[0][0]
cv2.putText(orig, "Label: {}, {:.2f}%".format(label, prob * 100),
(10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 0, 255), 2)
cv2.imshow("Classification", orig)
cv2.waitKey(0)
```

To see our pre-trained ImageNet networks in action, take a look at the next section.

VGGNet, ResNet, Inception, and Xception classification results

All ***updated*** examples in this blog post were gathered TensorFlow 2.2. Previously this blog post used **Keras >= 2.0** and a

Flask URL Building

The url_for() function is used to build a URL to the specific function dynamically. The first argument is the name of the specified function, and then we can pass any number of keyword argument corresponding to the variable part of the URL.

This function is useful in the sense that we can avoid hard-coding the URLs into the templates by dynamically building them using this function.

Consider the following python flask script.

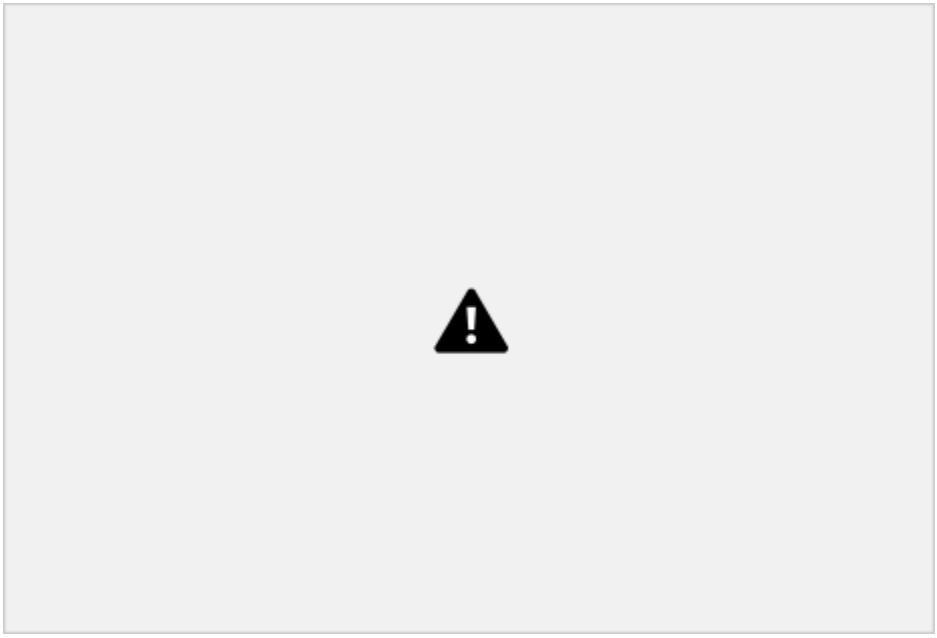
Example

```
1. from flask import *
2.
3. app = Flask(__name__) 4.
5. @app.route('/admin')
6. def admin():
7.      return
'admin' 8.
9. @app.route('/librarion')
10. def librarion():
11.      return
'librarion' 12.
13. @app.route('/student')
14. defstudent():
15.      return
'student' 16.
17. @app.route('/user/<name>')
```

```
18. def user(name):
19. if name == 'admin':
20. return redirect(url_for('admin'))
21. if name == 'librarion':
22. return redirect(url_for('librarion'))
23. if name == 'student':
24. return redirect(url_for('student'))
25. if__name___=='_main_':
26. app.run(debug = True)
```

The above script simulates the library management system which can be used by the three types of users, i.e., admin, librarion, and student. There is a specific function named user() which recognizes the user the redirect the user to the exact function which contains the implementation for this particular function.

x



For example, the URL http://localhost:5000/user/admin is redirected to the URL http://localhost:5000/admin, the URL localhost:5000/user/librarion, is redirected to the URL http://localhost:5000/librarion, the URL http://localhost:5000/user/student is redirected to the URL http://localhost/student.

Benefits of the Dynamic URL Building

1. It avoids hard coding of the URLs.
2. We can change the URLs dynamically instead of remembering the manually changed hard-coded URLs.
3. URL building handles the escaping of special characters and Unicode data transparently. 4. The generated paths are always absolute, avoiding unexpected behavior of relative paths in browsers.
5. If your application is placed outside the URL root, for example, in /myapplication instead of /, url_for() properly handles that for you.