

# C05M03S02- Building CNNs with Python and Keras

Deep Learning & Neural Networks Convolutional Neural Networks

1. Introduction
2. Building CNNs in Keras - MNIST
3. Comprehension - VGG16 Architecture
4. CIFAR-10 Classification with Python - I
5. CIFAR-10 Classification with Python - II
6. CIFAR-10 Classification with Python - III
7. Summary
8. Graded Questions

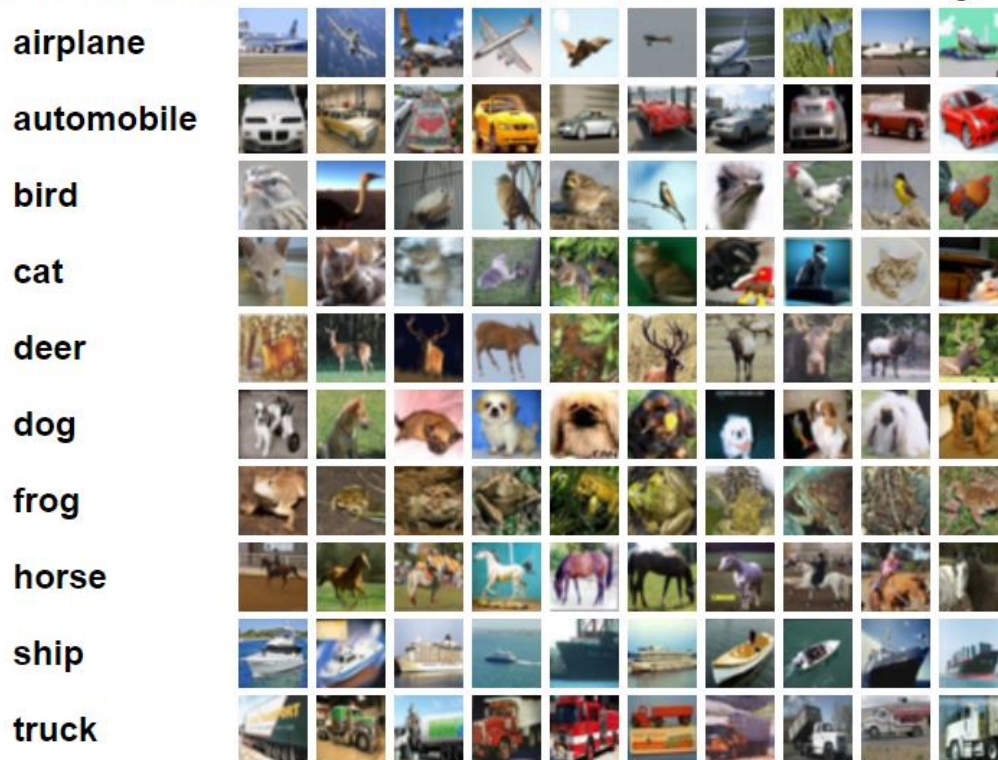
## Introduction

This session covers **training CNNs using Python + Keras**. More hands-on and a lot of time reading and modifying Python + Keras code and train models on GPUs.

To get started with the syntax and the process of building CNNs in Keras, we will first use the MNIST dataset. We will also learn to compute the number of parameters, output sizes etc. of each layer of a network.

Throughout the rest of the session, we will use the **CIFAR-10** dataset which has 60000 (32 x 32) colour images of 10 classes as shown below. In these exercises, we will also experiment with some hyperparameters of CNNs.

Here are the classes in the dataset, as well as 10 random images from each:



**CIFAR-10 dataset**

[In this session](#)

Define own convolutional layers in Keras and train those layers on the CIFAR-10 dataset. You will implement everything on a GPU.

- Get familiar with CNNs in Keras: The MNIST dataset
- Setting up your notebook on a GPU
- Conduct experiments with the **CIFAR-10** dataset:
  - Build a base model using the CIFAR-10 dataset
  - Experiment with hyperparameters and draw observations

## Prerequisites

There are no prerequisites for this session other than knowledge of the previous session.

## Building CNNs in Keras - MNIST

This segment will build CNNs in Keras. Specifically, to build and train a **CNN on the MNIST dataset** to classify the digits into one of the ten classes (0-9).

This is a **text only page** (with an IPython notebook) whose objective is to build CNNs in Keras. Next few segments cover some more experiments with CNNs using Python + Keras using the CIFAR-10 dataset.

No need to download the dataset separately, it can be downloaded from Keras directly (as done in the notebook).

Please **run this notebook locally**, not on a GPU. You will use the GPU in the upcoming segments. Make sure you understand the section '**Understanding Model Summary**' in the notebook well - it will be required to solve the questions on the next page.

[Building CNNs in Keras - MNIST](#)

[file downloadDownload](#)

In the next segment, you will test your understanding of the concepts covered in the notebook by solving some questions on the VGG-16 architecture.

## Comprehension - VGG16 Architecture

Dissect each layer of the VGG-16 architecture.

The VGG-16 was trained on the ImageNet challenge (ILSVRC) **1000-class classification** task. The network takes a (224, 224, 3) RGB image as the input. The '16' in its name comes from the fact that the network has 16 layers with trainable weights - **13 convolutional layers** and **3 fully connected** ones (the VGG team had tried many other configurations, such as the VGG-19, which is also quite popular).

The architecture is given in the table below (taken [from the original paper](#)). Each column in the table (from A-E) denotes an architecture the team had experimented with. In this discussion, we will refer only to **column D** which refers to **VGG-16** (column E is VGG-19).

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 <b>LRN</b>	conv3-64 <b>conv3-64</b>	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 <b>conv3-128</b>	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 <b>conv1-256</b>	conv3-256 conv3-256 <b>conv3-256</b>	conv3-256 conv3-256 conv3-256 <b>conv3-256</b>
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 <b>conv1-512</b>	conv3-512 conv3-512 <b>conv3-512</b>	conv3-512 conv3-512 conv3-512 <b>conv3-512</b>
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 <b>conv1-512</b>	conv3-512 conv3-512 <b>conv3-512</b>	conv3-512 conv3-512 conv3-512 <b>conv3-512</b>
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

### VGG-16

The convolutional layers are denoted in the table as conv<size of filter>-<number of filters>. Thus, conv3-64 means 64 (3, 3) square filters. Note that all the conv layers in VGG-16 use (3, 3) filters and that the number of filters increases in powers of two (64, 128, 256, 512).

In all the convolutional layers, the same **stride length of 1 pixel** is used with a **padding of 1 pixel** on each side, thereby preserving the spatial dimensions (height and width) of the output.

After every set of convolutional layers, there is a **max pooling** layer. All the pooling layers in the network use a **window of 2 x 2 pixels with stride 2**. Finally, the output of the last pooling layer is **flattened** and fed to a **fully connected (FC)** layer with 4096 neurons, followed by another FC layer of 4096 neurons, and finally to a 1000-softmax output. The softmax layer uses the usual cross-entropy loss. All layers apart from the softmax use the ReLU activation function.

The number of parameters and the output size from any layer can be calculated as demonstrated in the MNIST notebook on the previous page. For example, the first convolutional layer takes a (28, 28, 3) image as the input and has 64 filters of size (3, 3, 3). Note that the **depth of a filter** is always **equal to the number of channels** in the input which it convolves. Thus, the first convolutional layer has  $64 \times 3 \times 3 \times 3$  (weights) + 64 (biases) = 1792 trainable parameters. Since stride and padding of 1 pixel are used, the output spatial size is preserved, and the output will be (28, 28, 64).

Now answer the following questions (you will need a calculator). Keep track of the number of channels at each layer. Don't forget to add the biases.

<p><b>Conv Layer-2</b></p> <p>The output of the first convolutional layer is (224, 224, 64), i.e. 64 feature maps of size (224, 224). The second conv layer uses 64 filters of size (3, 3, 64). Note that the number of channels in the filters (64) is implicit since the filters have to convolve a tensor of 64 channels.</p> <p>The number of parameters in the second conv layer are:</p> <div><div><input checked="" type="radio"/> 36864</div><div>Feedback : Don't forget to add the biases.</div></div> <div><div><input type="radio"/> 36928</div><div>Feedback : The 64 (3, 3, 64) filters have <math>64 \times 3 \times 3 \times 64</math> (weights) + 64 (biases).</div></div> <div><div><input type="radio"/> 1792</div></div>
--

<p><b>Conv Layer-3 Output</b></p> <p>The output from the first pooling layer, (112, 112, 64), is fed to the third conv layer. The output of the third convolutional layer is:</p> <div> <input checked="" type="radio"/> (112, 112, 128) ✓         </div> <p>Q Feedback : Since all the convolutions are with stride 1 and padding 1, the size (height x width) is maintained. The third layer has 128 filters each of which will produce a 112 x 112 feature map.</p> <div> <input type="radio"/> (112, 112)         </div> <div> <input type="radio"/> (56, 56, 128)         </div>	<p><b>The First FC Layer</b></p> <p>Let's now come to the latter part of the network. The output from the last (13th) convolutional layer is of size (14, 14, 512) which is fed to a max pooling layer to give a (7, 7, 512) output. The output from the max pooling layer is then fed to a fully connected layer (FC) with 4096 neurons (after flattening).</p> <p>The number of trainable parameters in this FC layer is:</p> <div> <input checked="" type="radio"/> 102764544 ✗         </div> <p>Q Feedback : Don't forget the biases.</p> <div> <input type="radio"/> 102764544 ✓         </div> <p>Q Feedback : The output of the pooling layer, after flattening, will be a vector of length 7*7*512. Thus, the FC layer will have 7*7*512*4096 (weights) + 4096 (biases).</p>
<p><b>Shrinkage in VGG-16</b></p> <p>In the VGG-16 network, the size of the output is shrunk (i.e. the height x width of the input):</p> <div> <input type="radio"/> By only the convolutional layers         </div> <div> <input checked="" type="radio"/> By only the pooling layers ✓         </div> <p>Q Feedback : Since all the conv layers use a stride and padding of 1 with a (3, 3) filter, the spatial size is preserved in all the convolutional layers (only the depth increases). The height and width is reduced only by the pooling layers.</p> <div> <input type="radio"/> By both convolutional and pooling layers         </div>	

Total number of trainable parameters in the VGG-16 is about **138 million** (138,357,544 exactly), which is enormous. Some of the recent architectures (such as ResNet etc.) have achieved much better performance with far less number of parameters.

In the next few segments, the professor will demonstrate some experiments with various CNN hyperparameters on the CIFAR-10 dataset.

## CIFAR-10 Classification with Python - I

Next few segments will train various CNN networks on the [CIFAR-10 dataset](#). It has 10 classes of 60,000 RGB images each of size (32, 32, 3). The 10 classes are aeroplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. A similar dataset is the CIFAR-100 dataset which has 100 classes.

### Getting Started with Paperspace GPU and Google Cloud

Use a GPU to run the CIFAR-10 notebooks (running each notebook locally will take 2-3 hours, on a GPU it will take 8-10 minutes). The instructions for [setting up the notebooks on GPUs are explained in this video](#).

In case Paperspace is down, use Google Colab. [Learn how to get started with Google Colab here](#). Use Paperspace as the primary cloud provider and Google Colab only when Paperspace is down.

### CIFAR-10 Experiments

Coming few lectures will experiment with some hyperparameters and architectures and draw insights from the results. Some hyperparameters we will play with are:

- Adding and removing dropouts in convolutional layers
- Batch Normalization (BN)
- L2 regularisation
- Increasing the number of convolution layers
- Increasing the number of filters in certain layers

**Experiment - I:** Using dropouts after conv and FC layers

In the first experiment, we will use **dropouts** both after the convolutional and fully connected layers.

#### Download – Notebook

[https://cdn.upgrad.com/UpGrad/temp/50032fac-8cc9-4bce-842b-d5e768a40c01/1.+Cifar\\_10\\_with\\_dropout\\_without\\_BN.ipynb](https://cdn.upgrad.com/UpGrad/temp/50032fac-8cc9-4bce-842b-d5e768a40c01/1.+Cifar_10_with_dropout_without_BN.ipynb)

The results of the experiment are as follows:

**Experiment - I:** Dropouts After Conv and FC layers

- Training accuracy = 84%, validation accuracy = 79%

In the next few segments, we will conduct some more experiments (without dropouts, using batch normalisation, adding more convolutional layers etc) and compare the results.



## CIFAR-10 Classification with Python - II

In the first experiment (using dropouts after both convolutional and FC layers), we got training and validation accuracies of about 84% and 79% respectively. Let's now run three different experiments as mentioned below and compare the performance:

**Experiment - II:** Remove dropouts after the convolutional layers (but retain them in the FC layer). Also, use **batch normalization** after every convolutional layer.

Batch normalisation (BN) normalises the outputs from each layer with the mean and standard deviation of the batch. Quickly [revisit the lectures on BN here](#).

**Experiment - III:** Use batch normalization and dropouts after every convolutional layer. Also, retain dropouts in the FC layer.

**Experiment - IV:** Remove dropouts after convolutional layers and use **L2 regularization** in the FC layer. Retain dropouts in FC.


[https://cdn.upgrad.com/UpGrad/temp/dabeb6ba-e728-4520-b5d4-683749f0cf25/2.+Cifar\\_10\\_Notebook\\_with\\_BN\\_without\\_dropout.ipynb](https://cdn.upgrad.com/UpGrad/temp/dabeb6ba-e728-4520-b5d4-683749f0cf25/2.+Cifar_10_Notebook_with_BN_without_dropout.ipynb)  
[https://cdn.upgrad.com/UpGrad/temp/4d775251-0c23-4184-9300-450d94b46168/3.+Cifar\\_10\\_notebook.ipynb](https://cdn.upgrad.com/UpGrad/temp/4d775251-0c23-4184-9300-450d94b46168/3.+Cifar_10_notebook.ipynb)  
[https://cdn.upgrad.com/UpGrad/temp/3ddf0887-466e-4263-89d9-fa26ababa035/4.+Cifar10\\_12\\_notebook.ipynb](https://cdn.upgrad.com/UpGrad/temp/3ddf0887-466e-4263-89d9-fa26ababa035/4.+Cifar10_12_notebook.ipynb)

## CNN Experiments

The results of the experiments done so far are summarised below. Based on these, choose all the correct options:

- **Experiment - I** (Use dropouts after conv and FC layers, no BN):
  - Training accuracy = 84%, validation accuracy = 79%
- **Experiment - II** (Remove dropouts from conv layers, retain dropouts in FC, use BN):
  - Training accuracy = 98%, validation accuracy = 79%
- **Experiment - III** (Use dropouts after conv and FC layers, use BN):
  - Training accuracy = 89%, validation accuracy = 82%
- **Experiment - IV** (Remove dropouts from conv layers and use L2 + dropouts in FC, use BN):
  - Training accuracy = 94%, validation accuracy = 76%.


☐ The ideal configuration (from the ones tried so far) is to use dropouts after both conv and FC layers without BN

☒ The ideal configuration (from the ones tried so far) is to use dropouts after both conv and FC layers with BN 

💡 Feedback :  
This corresponds to experiment-III which has given the best results so far.

☒ Removing dropouts after the conv layers affects the performance adversely 

💡 Feedback :  
In experiments II and IV, we had removed the dropouts after the conv layers, and the performance reduced drastically (the model overfits).

☐ Using BN (keeping other things constant) improves the performance significantly 

💡 Feedback :  
Compare experiments I and IV - using BN improves both the training and validation accuracies.

## CIFAR-10 Classification with Python - III

Let's continue our experiments further. We have learnt that dropouts are pretty useful, batch normalisation somewhat helps improve performance, and that L2 regularisation is not very useful on its own (i.e. without dropouts).

Let's now conduct an experiment with all these thrown in together. After this experiment, let's conduct another one to test whether adding a new convolutional layer helps improve performance.

**Experiment-V:** Dropouts after conv layer, L2 in FC, use BN after convolutional layer

**Experiment-VI:** Add a **new convolutional layer** to the network. By a 'convolutional layer', we are referring to a convolutional unit with two sets of Conv2D layers with 128 filters each (we are abusing the terminology a bit here). The code for the additional conv layer is shown below.

```
an additional conv unit
model.add(Conv2D(128, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(Conv2D(128, (3, 3)))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
```

[Cifar10 12 dropout notebook](#)

[Cifar10 morelayer notebook](#)

The results of these two experiments are summarised below.

**Experiment-V:** Dropouts after conv layer, L2 in FC, use BN after convolutional layer

- Train accuracy = 86%, validation accuracy = 83%

**Experiment-VI:** Add a new convolutional layer to the network

- Train accuracy = 89%, validation accuracy = 84%

The additional convolutional layer boosted the validation accuracy marginally, but due to increased depth, the training time increased.

### Adding Feature Maps

In the previous experiment, we tried to increase the capacity of the model by adding a convolutional layer. Let's now try adding more feature maps to the same architecture.

**Experiment - VII:** Add more feature maps to the conv layers: from 32 to 64 and 64 to 128.

[https://cdn.upgrad.com/UpGrad/temp/31f5db43-a84d-4d6d-964b-dbf41259047c/7.+Cifar10\\_feature\\_map.ipynb](https://cdn.upgrad.com/UpGrad/temp/31f5db43-a84d-4d6d-964b-dbf41259047c/7.+Cifar10_feature_map.ipynb)

The results of our final experiment are mentioned below.

**Experiment-VII:** Add more feature maps to the convolutional layers to the network

- Train accuracy = 92%, validation accuracy = 84%

On adding more feature maps, the model tends to overfit (compared to adding a new convolutional layer). This shows that the task requires learning to extract more (new) abstract features, rather than trying to extract more of the same features.

## Summary

This session covered building and training CNNs in Keras and experimented some hyperparameters of the model. You also practised manually computing the number of parameters, output sizes etc. of CNN-based architectures.

Based on these experiments, we saw that the performance of CNNs depends heavily on multiple hyperparameters –

- the number of layers,
- number of feature maps in each layer,
- the use of dropouts,
- batch normalisation,

etc.

Thus, it is advisable to first fine-tune your model hyperparameters by conducting lots of experiments. Only when you are convinced that you have found the right set of hyperparameters you should train the model with a larger number of epochs (since almost always the amount of time and computing power is limited).

In the next session, you will study the architectures of some popular deep convolutional networks, learn to train CNNs in Python + Keras, and use large pre-trained networks for your own tasks using transfer learning.

<p><b>Comprehension</b></p> <p>If the spatial dimensions (width and height) of the output going into the third layer are the same as the input from the previous layer, what can be the possible values of stride 's1' and padding 'p1'?</p> <p><input type="radio"/> stride 1, padding 1</p> <p><input type="radio"/> stride 2, padding 2</p> <p><input checked="" type="radio"/> stride 1, padding 2 ✓</p> <p>Q Feedback: Calculate the output using <math>((n+2p-k)/s + 1)</math>. With <math>s=1, p=2</math>, the output is <math>(512 + 4 - 5)/1 + 1 = 512</math>.</p> <p><input type="radio"/> stride 2, padding 1</p>	<p><b>Comprehension</b></p> <p>The 8th layer is named layer 'T'. Which of the following types could be the layer 'T'?</p> <p><input type="radio"/> Convolution</p> <p><input type="radio"/> Pooling</p> <p><input checked="" type="radio"/> Flatten ✓</p> <p>Q Feedback: The 'Flatten' layer connects the convolutional layer to the fully connected layer by flattening the multidimensional tensor output from the conv layer to a long vector.</p> <p><input type="radio"/> Fully connected</p>
<p><b>Comprehension</b></p> <p>What is the output from the last max pooling layer (layer 7) assuming that the width and the height do not change after the convolution operation in step-2?</p> <p><input type="radio"/> 256x256x32</p> <p><input checked="" type="radio"/> 128x128x64 ✓</p> <p>Q Feedback: After two pooling operations (starting from the starting <math>512 \times 512</math>), the width and the height will reduce 2 times, i.e. from 512 to 256 (in the first max pooling layer) and from 256 to 128 (in the second max pooling layer).</p>	<p><b>Comprehension</b></p> <p>What is the value of 'F' in the last layer?</p> <p><input type="radio"/> 1000</p> <p><input checked="" type="radio"/> 3 ✓</p> <p>Q Feedback: Since we are classifying an image into 3 classes, it has to have 3 neurons.</p>

### Comprehension

Calculate the total number of trainable parameters in layer-3 (the conv layer with 32 3x3 filters)?

☒ 9248



Feedback :

The output from the previous layer is (512, 512, 32), so each filter is of size (3, 3, 32). The number of parameters is thus 32 filters \* 3\*3\*32 (weights) + 32 (biases) = 9248.

☐ 10,272

☐ 9216

☐ 320