

# C05M03S03- CNN Architectures and Transfer Learning

Deep Learning & Neural Networks Convolutional Neural Networks

1. Introduction
2. Overview of CNN Architectures
3. AlexNet and VGGNet
4. GoogleNet
5. Residual Net
6. Introduction to Transfer Learning
7. Use Cases of Transfer Learning
8. Transfer Learning With Pre-Trained CNNs
9. Practical Implementation of Transfer Learning
10. Transfer Learning in Python
11. An Analysis of Deep Learning Models - I
12. An Analysis of Deep Learning Models - II
13. Summary
14. Graded Questions

## Introduction

Previous session analysed the architecture of VGGNet in detail. This session will cover some other famous architectures which had achieved significantly better performance in the ImageNet competition over their predecessors. Specifically, the architectures of **AlexNet**, **GoogleNet** and **ResNet**.

Use these pre-trained models for own problems using what is called **Transfer Learning**. Finally, we will conclude this session by analysing the performance (not just the accuracy, but efficiency as well) of various popular CNNs for practical deployment purposes, a study recently published in the paper "**An analysis of Deep Neural Network Models for Practical Applications** " by A Canziani.

In this Session:

- CNN architectures: AlexNet, GoogleNet, ResNet
- Transfer Learning
- Analysis of Deep Neural Network Models

Prerequisites

There are no prerequisites for this session other than knowledge of the matrix multiplication and the previous session on CNN architecture.

## Overview of CNN Architectures

Overview of some of the most popular CNN architectures which have set the benchmark for state-of-the-art results in computer vision tasks. The acid test for almost CNN-based architectures has been the [ImageNet Large Scale Visual Recognition Competition](#), or simply **ImageNet**. The dataset contains roughly 1.2 million training images, 50,000 validation images, and 150,000 testing images of about 1000 classes.

We will discuss the following architectures in this session:

- AlexNet
- VGGNet
- GoogleNet
- ResNet

Important points:

- **Depth** of the state-of-the-art neural networks has been **steadily increasing** (from AlexNet with 8 layers to ResNet with 152 layers).
- Developments in neural net architectures were made possible by **significant advancements in infrastructure**. Many of these networks were trained on multi GPUs in a distributed manner.
- Since these networks have been trained on millions of images, they are good at **extracting generic features** from a large variety of images. Thus, they are now commonly being used as commodities by deep learning practitioners around the world.

You will learn to use large pre-trained networks in the next section on **transfer learning**. In the next segment, we will study the architectures of AlexNet, VGGNet and GoogleNet.

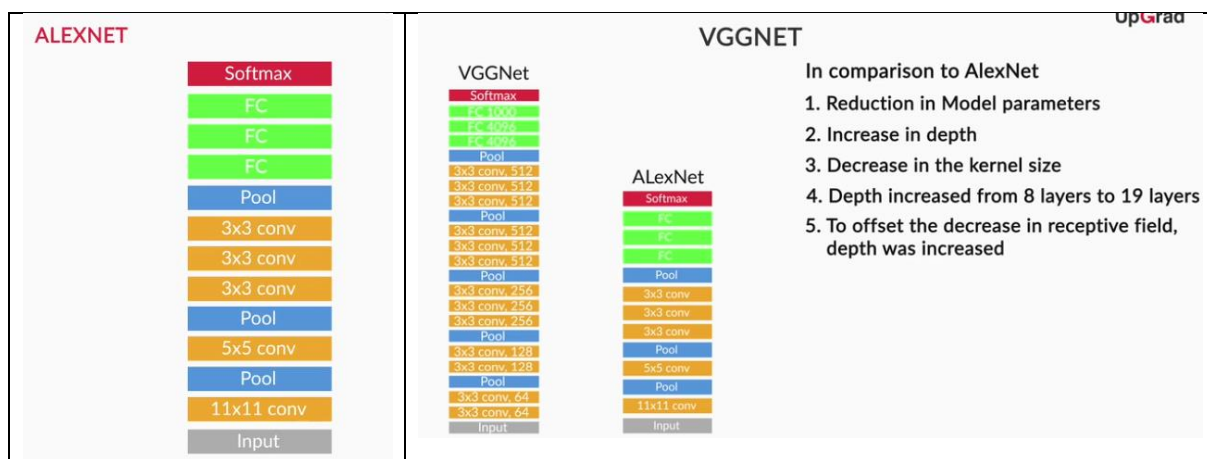
## AlexNet and VGGNet

In this session, we will briefly look into the architectures of **AlexNet** and **VGGNet**.

**AlexNet** was one of the very first architectures to achieve extraordinary results in the ImageNet competition (with about a 17% error rate). It had used **8 layers** (5 convolutional and 3 fully connected). One distinct feature of AlexNet was that it had used various **kernels of large sizes** such as (11, 11), (5, 5), etc. Also, AlexNet was the first to use dropouts, which were quite recent back then.

We are already familiar with **VGGNet** which has used **all filters of the same size** (3, 3) and had more layers (The VGG-16 had 16 layers with trainable weights, VGG-19 had 19 layers etc.).

The VGGNet had succeeded AlexNet in the ImageNet challenge by reducing the error rate from about 17% to less than 8%. Let's compare the architectures of both the nets.



There are some other important points to note about AlexNet which are summarised below. We highly [recommend you to go through the AlexNet paper](#) . Because of lack of good computing hardware, it was trained on **smaller GPUs** (3 GBs of RAM). Thus, the training was **distributed across two GPUs** in parallel (figure shown below). AlexNet was also the first architecture to use the ReLU activation heavily.

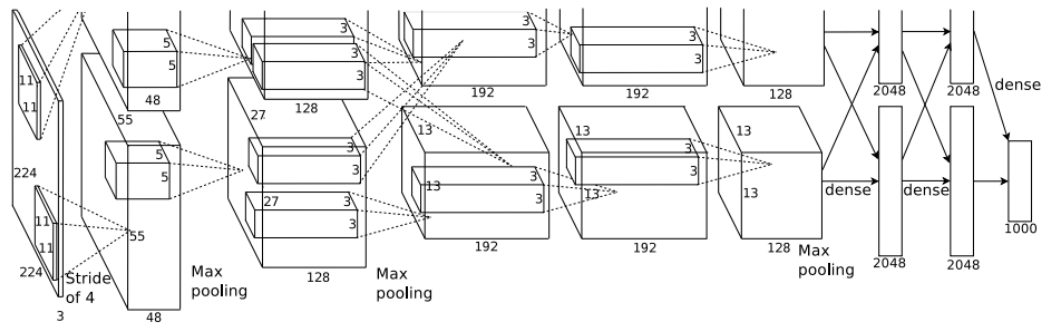


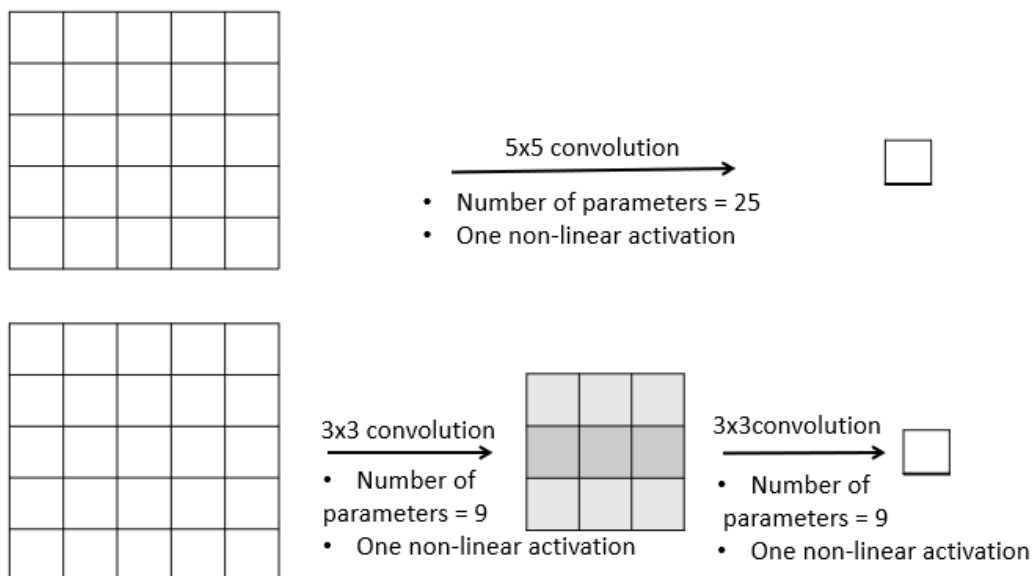
Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

## AlexNet

### Comprehension - Effective Receptive Field

The key idea in moving from AlexNet to VGGNet was to **increase the depth** of the network by using **smaller filters**. Let's understand what happens when we use a smaller filter of size (3, 3) instead of larger ones such as (5, 5) or (7, 7).

Say we have a 5 x 5 image, and in two different convolution experiments, we use two different filters of size (5, 5) and (3, 3) respectively.



### 5x5 convolution

In the first convolution, the (5, 5) filter produces a feature map with a single element (note that the convolution is followed by a non-linear function as well). This filter has 25 parameters.

In the second case with the (3, 3) filter, two successive convolutions (with stride=1, no padding) produce a feature map with one element.

We say that the stack of two (3, 3) filters has the same **effective receptive field** as that of one (5, 5) filter. This is because both these convolutions produce the same output (of size 1 x 1 here) whose receptive field is the same 5 x 5 image.

With a smaller (3, 3) filter, we can make a deeper network with **more non-linearities** and **fewer parameters**. In the above case:

- The (5, 5) filter has 25 parameters and one non-linearity
- The (3, 3) filter has 18 (9+9) parameters and two non-linearities.

Since VGGNet had used smaller filters (all of 3 x 3) compared to AlexNet (which had used 11 x 11 and 5 x 5 filters), it was able to use a higher number of non-linear activations with a reduced number of parameters.

In the next segment, we will briefly study **GoogleNet** which had outperformed VGGNet.

#### Effective Receptive Field

A (7, 7) filter has the same effective receptive field as (assuming zero padding and stride length 1):

☐ Two (3, 3) filters

☐ Two (5, 5) filters

☒ Three (3, 3) filters



#### Q Feedback :

Consider an  $(n, n)$  image. A (7, 7) filter will result in an  $(n-6, n-6)$  output. Now consider some convolutions with a (3, 3) filter. The first convolution will produce an  $(n-2, n-2)$  output, the second will produce an  $(n-4, n-4)$  output, and the third will produce an  $(n-6, n-6)$  output.

☐ Two (5, 5) filters

## Additional Readings

We strongly recommend reading the AlexNet and VGGNet papers provided below.

1. [The AlexNet paper, Alex Krizhevsky et. al.](#)
2. [The VGGNet paper, Karen Simonyan et. al.](#)

## GoogleNet

After VGGNet, the next big innovation was the **GoogleNet** which had won the ILSVRC'14 challenge with an error rate of about 6.7%.

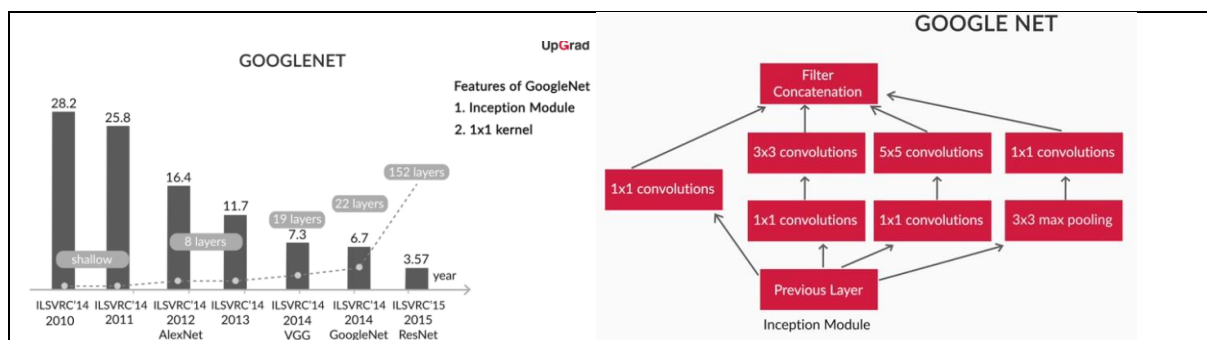
Unlike the previous innovations, which had tried to increase the model capacity by adding more layers, reducing the filter size etc. (such as from AlexNet to VGGNet), GoogleNet had increased the depth using a new type of convolution technique using the **Inception module**.

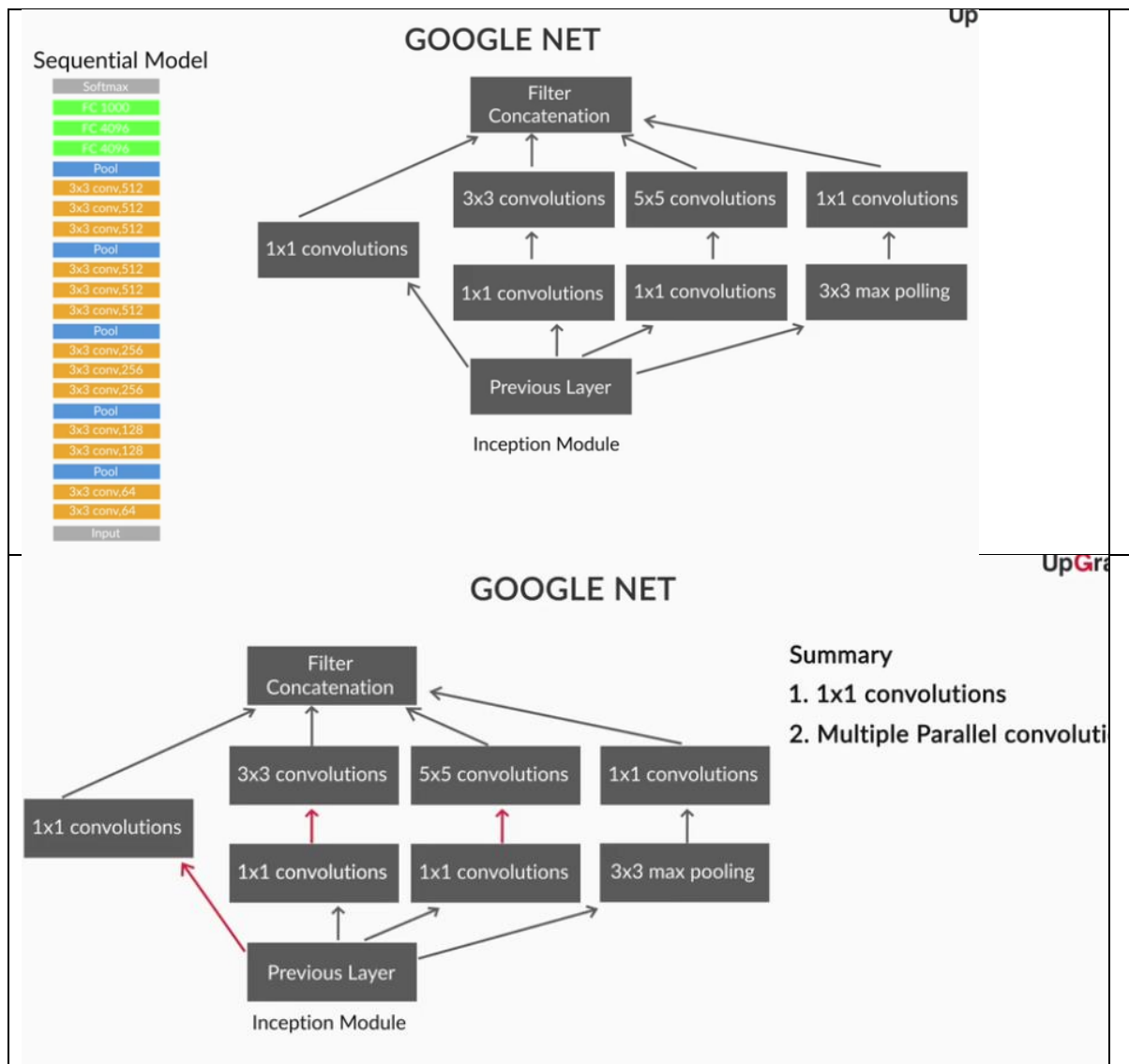
The module derives its name from a previous paper by Lin et al and this meme popular in the deep learning community:



Inception

Let's study the key features of GoogleNet architecture.





To summarise, some important features of the GoogleNet architecture are:

- Inception modules stacked on top of each other, total 22 layers
- Use of 1 x 1 convolutions in the modules
- Parallel convolutions by multiple filters (1x1, 3x3, 5x5)
- Pooling operation of size (3x3)
- No FC layer, except for the last softmax layer for classification
- Number of parameters reduced from 60 million (AlexNet) to 4 million

Details on why GoogleNet and inception module work well are beyond scope of this course.

In the next segment, we will look at the architecture of **ResNet**.

### Additional Reading

1. [The GoogleNet, Christian Szegedy et al](#)



## Residual Net

Until about 2014 (when the GoogleNet was introduced), the most significant improvements in deep learning had appeared in the form of **increased network depth** - from the AlexNet (8 layers) to GoogleNet (22 layers). Some other networks with around 30 layers were also introduced around that time.

Driven by the significance of depth, a team of researchers asked the question: *Is learning better networks as easy as stacking more layers?*

The team experimented with substantially deeper networks (with hundreds of layers) and found some counterintuitive results (shown below). In one of the experiments, they found that a 56-layered convolutional net had a higher training (and test) error than a 20-layered net on the CIFAR-10 dataset.

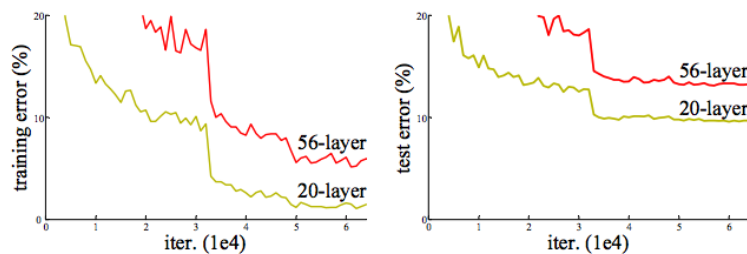


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

### ResNet

Analyse the results in the plot above and list down at least 1-2 possible explanations for them.

## Deeper Nets

We mentioned that a team of deep learning researchers, in their experiments with deeper networks, found that a 56-layer conv net performed worse than a 20-layered net. The results from their experiments are shown below. What can be a possible reason for these results?

This is a poll question. Choose an option and compare your answer with your peers.

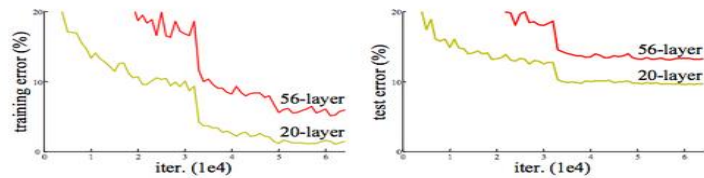


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

☐ The deeper network overfitted

42%

☐ The deeper network was harder to train because of infrastructural inadequacies

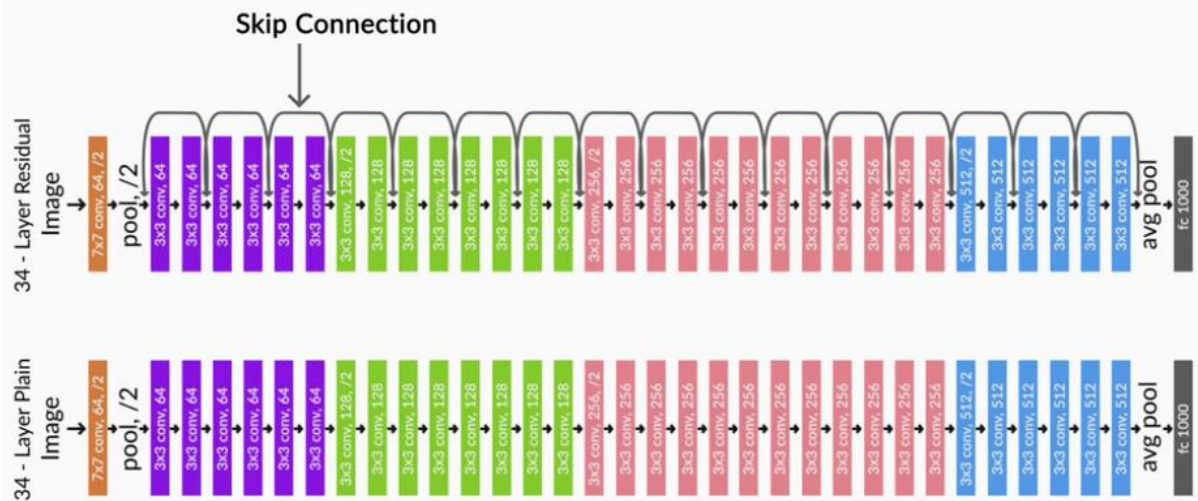
14%

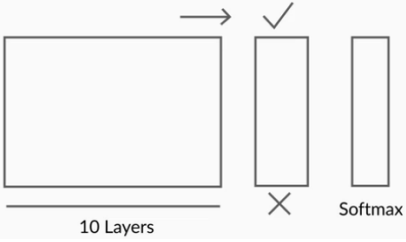
☒ The deeper network was harder to train because of conceptual problems such as exploding / vanishing gradients

44%

Only 100 people have taken this poll from your cohort. The results would be updated as we get more responses.

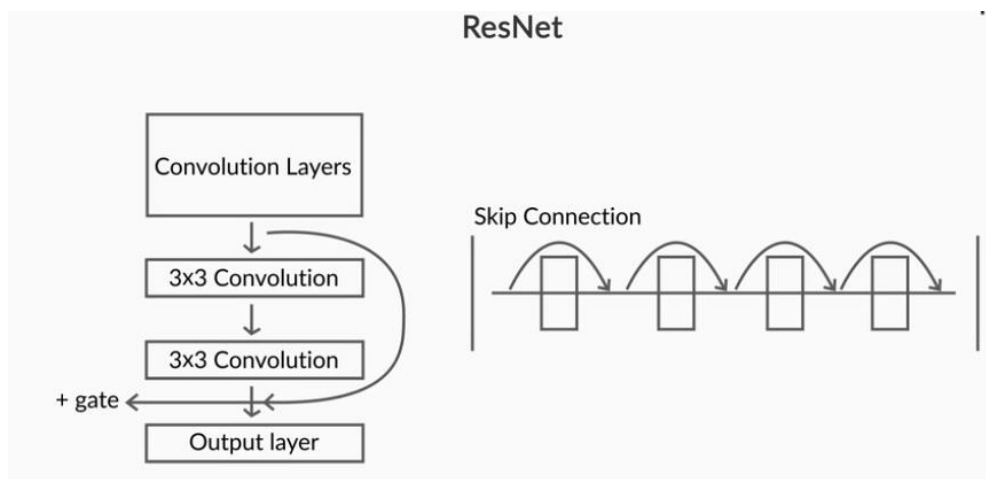
## ResNet



<p style="text-align: center;"><b>ResNet</b></p>  <p style="text-align: center;">10 Layers</p>	<p>Try to ensure that additional layer added does not reduce the accuracy.</p> <p><b>Key motivator</b> for the ResNet architecture was the observation that, empirically, adding more layers was not improving the results monotonically. This was counterintuitive because a network with <math>n + 1</math> layers should be able to learn <i>at least</i> what a network with <math>n</math> layers could learn, plus something more.</p>
---	--

The ResNet team ([Kaiming He et al](#)) came up with a novel architecture with **skip connections** which enabled them to train networks as deep as **152 layers**. The ResNet achieved ground breaking results across several competitions - a 3.57% error rate on the ImageNet and the first position in many other ILSVRC and [COCO object detection](#) competitions.

Let's look at the basic mechanism, the **skip connections** or **residual connections**, which enabled the training of very deep networks.



**Skip connection mechanism** was the key feature of ResNet which enabled training of very deep networks. Some other key features of ResNet are summarised below.

- ILSVRC'15 classification winner (3.57% top 5 error)
- 152 layer model for ImageNet
- Has other variants also (with 35, 50, 101 layers)
- Every 'residual block' has two 3x3 convolution layers
- No FC layer, except one last 1000 FC softmax layer for classification
- Global average pooling layer after the last convolution
- Batch Normalization after every convolution layer
- SGD + momentum (0.9)
- No dropout used

In the next few segments, you will learn how to use these large pre-trained networks to solve your own deep learning problems using the principles of **transfer learning**.

Additional Reading

1. [The Residual Net, Kaiming He et al](#)

## Introduction to Transfer Learning

So far, we have discussed multiple CNN based networks which were trained on millions of images of various classes. The ImageNet dataset itself has about 1.2 million images of 1000 classes.

However, what these models have 'learnt' is not confined to the ImageNet dataset (or a classification problem). In an earlier session, we had discussed that CNNs are basically **feature-extractors**, i.e. the convolutional layers learn a representation of an image, which can then be used for any task such as classification, object detection, etc.

This implies that the models trained on the ImageNet challenge have learnt to extract features from a wide range of images. Can we then **transfer this knowledge** to solve some other problems as well?

Thus, **transfer learning** is the practice of reusing the skills learnt from solving one problem to learn to solve a new, related problem. Before diving into how to do transfer learning, let's first look at some practical reasons to do transfer learning in the first place.

To summarise, some practical reasons to use transfer learning are:

- Data abundance in one task and data crunch in another related task.
- Enough data available for training, but lack of computational resources

An example of the first case is this - say you want to build a model (to be used in a driverless car to be driven in India) to classify 'objects' such as a pedestrian, a tree, a traffic signal, etc. Now, let's say you don't have enough labelled training data from Indian roads, but you can find a similar dataset from an American city. You can try training the model on the American dataset, take those learned weights, and then train further on the smaller Indian dataset.

Examples of the second use case are more common - say you want to train a model to classify 1000 classes, but don't have the infrastructure required. You can simply pick up a trained VGG or ResNet and train it a little more on your limited infrastructure. You will implement such a task in Keras shortly.

In the next segment, we will see some other use cases where we can use transfer learning.

## Use Cases of Transfer Learning

Let's continue our discussion on use cases of transfer learning using some examples from **natural language processing**.

Let's revisit the example of **document summarization**. If you want to do document summarisation in some other language, such as Hindi, you can take the following steps:

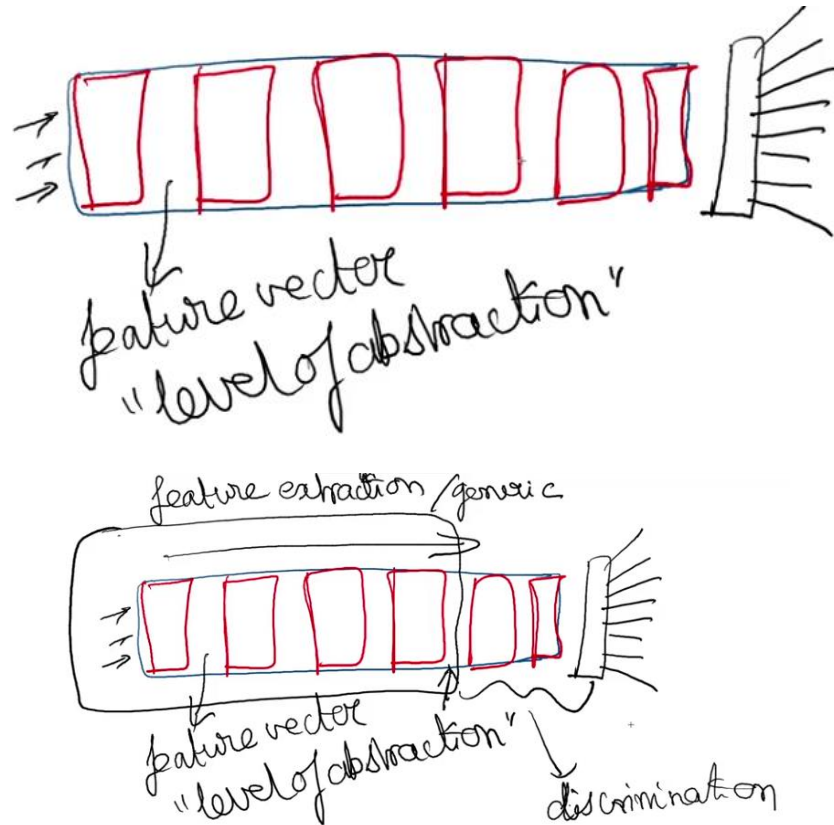
- Use word embeddings in English to train a document summarization model (assuming a significant amount of data in English is available)
- Use word embeddings of another language such as Hindi (where you have a data crunch) to tune the English summarization model

Let's now summarise the main idea of transfer learning.

In the next segment, you will see some common use cases of transfer learning applied to computer vision tasks.

## Transfer Learning with Pre-Trained CNNs

For most computer vision problems, you can usually better off using a pre-trained model such as AlexNet, VGGNet, GoogleNet, ResNet etc. Let's study how exactly one should go about doing this.



Thus, the initial layers of a network extract the basic features, the latter layers extract more abstract features, while the last few layers are simply discriminating between images.

In other words, the initial few layers are able to **extract generic representations of an image** and thus can be used for any general image-based task. Let's see some examples of tasks we can use transfer learning for.

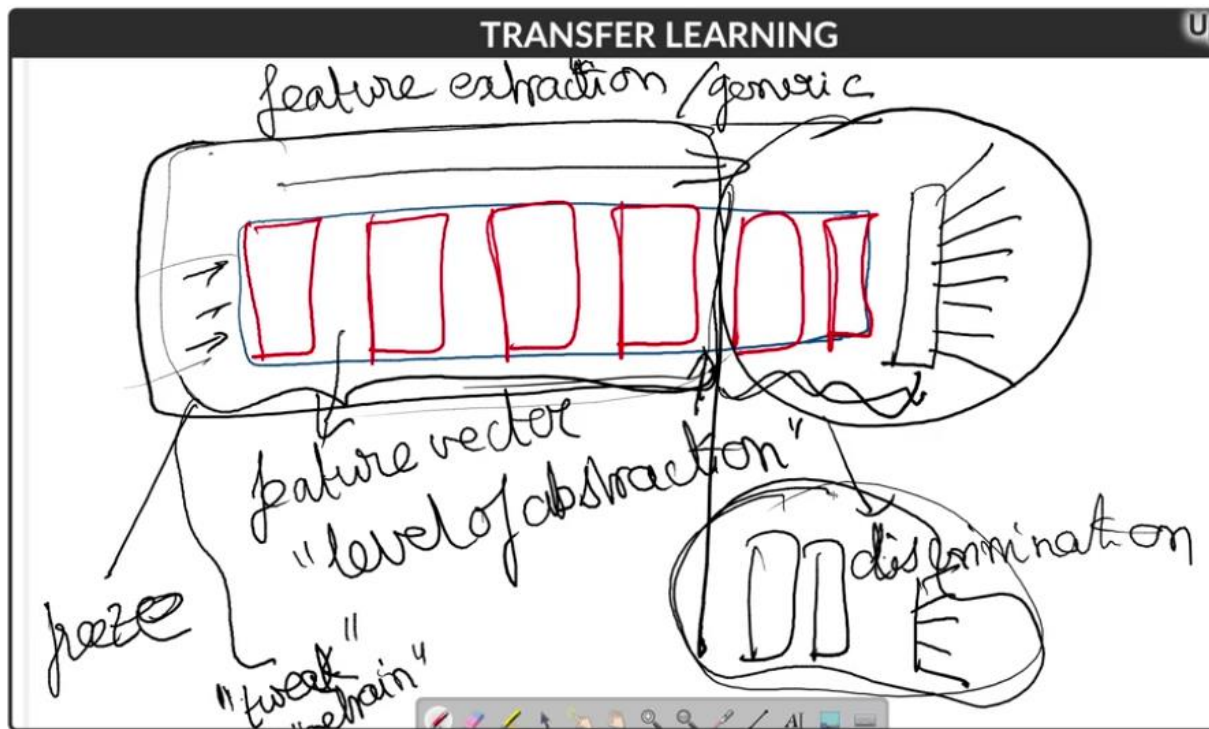
Thus, transfer learning is not just limited to image classification but can be extended to a wide variety of tasks. In the next segment, you will learn how to train pre-trained models for specific purposes.

## Practical Implementation of Transfer Learning

There are two main ways of using pre-trained nets for transfer learning:

- Freeze the (weights of) initial few layers and training only a few latter layers
- Retrain the entire network (all the weights) initialising from the learned weights

Let's look at these two techniques in detail.



Thus, you have the following two ways of training a pre-trained network:

1. **'Freeze'** the initial layers, i.e. use the same weights and biases that the network has learnt from some other task, remove the few last layers of the pre-trained model, add your own new layer(s) at the end and **train only the newly added layer(s)**.
2. **Retrain** all the weights **starting (initialising) from the weights and biases** that the net has already learnt. Since you don't want to unlearn a large fraction of what the pre-trained layers have learnt. So, for the initial layers, we will choose a low learning rate.

When you implement transfer learning practically, you will need to take some decisions such as how many layers of the pre-trained network to throw away and train yourself. Let's see how one can answer these questions.

To summarise:

- If the task is a lot similar to that the pre-trained model had learnt from, you can use most of the layers except the last few layers which you can retrain
- If you think there is less similarity in the tasks, you can use only a few initial trained weights for your task.

In the next segment, you will see a demonstration of Transfer Learning in Python + Keras.

## Transfer Learning in Python

Implement transfer learning in Python. For this implementation, we will use the [flower recognition dataset](#) from Kaggle. This dataset has around 4000 images from 5 different classes, namely daisy, dandelion, rose, sunflower and tulip.

### Running the Notebook on Paperspace GPU

We recommend that you use a GPU to run the transfer learning notebook (provided below). For running this notebook, you would require to create your Kaggle account and download the dataset from Kaggle directly. It takes approximately 20 minutes to run the code on GPU. The instructions for setting up the notebook and downloading the dataset are explained in the following videos:

- [Setting up notebook part-1](#)
- [Setting up notebook part-2](#)

You can download the notebook from [Transfer Learning Notebook](#).

To summarise, we conducted two transfer learning experiments. In the first experiment, we removed the last fully connected layers of ResNet (which had learnt how to classify the 1000 ImageNet images). Instead, we added our own pooling, fully connected and a 5-softmax layer and trained only those. Notice that we got very good accuracy in just a few epochs. In case we weren't satisfied with the results, we could modify this network further (add an FC layer, modify the learning rate, replace the global average pooling layer with max pool, etc.).

In the second experiment, we froze the first 140 layers of the model (i.e. used the pre-trained ResNet weights from layers 1-140) and trained the rest of the layers. Note that while updating the pre-trained weights, we should use a **small learning rate**. This is because we do not expect the weights to change drastically (we expect them to have learnt some generic patterns, and want to tune them only a little to accommodate for the new task).

In the next two segments, you will go through an interesting recent paper which compares various CNN architectures from an efficiency and deployment point of view.



## An Analysis of Deep Learning Models - I

In the past few years, the performance of CNN based architectures such as AlexNet, VGGNet, ResNet etc. has been steadily improving. But while **deploying deep learning models** in practice, you usually have to consider multiple other parameters apart from just accuracy.

For example, say you've built a mobile app which uses a conv net for real-time face detection. Since it will be deployed on smartphones, some of which may have low memory etc., you might be more worried about it working 'fast enough', rather than the accuracy.

In the next two segments, we will discuss a recent paper that appeared in 2017 - “[An Analysis of Deep Neural Network Models for Practical Applications](#)”. This paper compares the popular architectures on multiple metrics related to **resource utilisation** such as accuracy, memory footprint, number of parameters, operations count, inference time and power consumption.