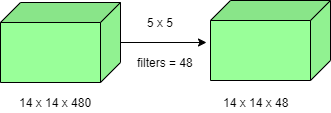
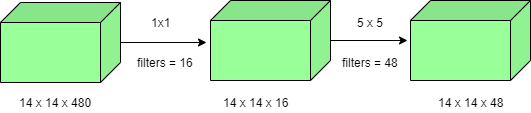
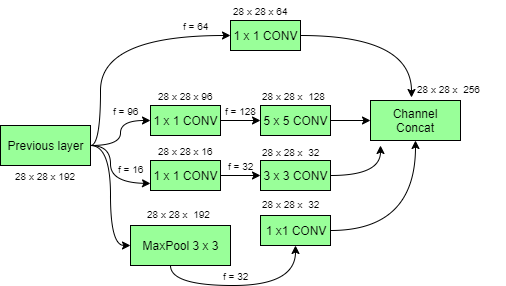
1. Using our own terms and diagrams, explain INCEPTIONNET ARCHITECTURE.

Inception net achieved a milestone in CNN classifiers when previous models were just going deeper to improve the performance and accuracy but compromising the computational cost. The Inception network, on the other hand, is heavily engineered. It uses a lot of tricks to push performance, both in terms of speed and accuracy. It is the winner of the ImageNet Large Scale Visual Recognition Competition in 2014, an image classification competition, which has a significant improvement over ZFNet (The winner in 2013), AlexNet (The winner in 2012) and has relatively lower error rate compared with the VGGNet (1st runner-up in 2014). The major issues faced by deeper CNN models such as VGGNet were:

* Although, previous networks such as VGG achieved a remarkable accuracy on the ImageNet dataset, deploying these kinds of models is highly computationally expensive because of the deep architecture.
* Very deep networks are susceptible to overfitting. It is also hard to pass gradient updates through the entire network.

Before digging into Inception Net model, it’s essential to know an important concept that is used in Inception network: **1 X 1 convolution:**A *1×1* convolution simply maps an input pixel with all its respective channels to an output pixel. *1×1* convolution is used as a dimensionality reduction module to reduce computation to an extent.

* For instance, we need to perform *5×5* convolution without using *1×1* convolution as below:Number of operations involved here is *(14×14×48) × (5×5×480) = 112.9M*
* Using 1×1 convolution:Number of operations for 1×1 convolution = *(14×14×16) × (1×1×480) = 1.5M* Number of operations for 5×5 convolution = *(14×14×48) × (5×5×16) = 3.8M* After addition we get, *1.5M + 3.8M = 5.3M*

**Which is immensely smaller than 112.9M !** Thus, 1×1 convolution can help to reduce model size which can also somehow help to reduce the overfitting problem. **Inception model with dimension reductions:**Deep Convolutional Networks are computationally expensive. However, computational costs can be reduced drastically by introducing a **1 x 1 convolution**. Here, the number of input channels is limited by adding an extra *1×1* convolution before the *3×3* and *5×5* convolutions. Though adding an extra operation may seem counter-intuitive but *1×1* convolutions are far cheaper than *5×5* convolutions. Do note that the *1×1* convolution is introduced after the max-pooling layer, rather than before. At last, all the channels in the network are concatenated together i.e. *(28 x 28 x (64 + 128 + 32 + 32)) = 28 x 28 x 256.***GoogLeNet Architecture of Inception Network:** This architecture has *22* layers in total! Using the dimension-reduced inception module, a neural network architecture is constructed. This is popularly known as **GoogLeNet (Inception v1)**. GoogLeNet has *9* such inception modules fitted linearly. It is *22* layers deep (*27*, including the pooling layers). At the end of the architecture, fully connected layers were replaced by a global average pooling which calculates the average of every feature map. This indeed dramatically declines the total number of parameters. Thus, Inception Net is a victory over the previous versions of CNN models. It achieves an accuracy of top-5 on ImageNet, it reduces the computational cost to a great extent without compromising the speed and accuracy.

*Note : What we just saw is an example of just one inception module with****dimension reduction****. The overall Inception network consists of many such inception modules stacked together.*

Another thing to make note of is that Inception network are also used to solve deep neural network problems. Let’s see what’s the problem :

When we design a deep neural network, we decide the number of layers it will contain and the number of neurons per layer so if the number of layers are more then it might result in following problems :

*1. The bigger our model is(more number of layers), the more it will be prone to the problem of overfitting (When number of input feature is large during training)*

*2. If the number of layers are large, the number of parameters will also increase and hence we’ll also be required to increase the computational resources only then it will be able to perform the computation on these parameters.*

So, rather then increasing the computational resource, we can use an Inception network which will minimize the computation costs while also increasing the depth and width of the network.

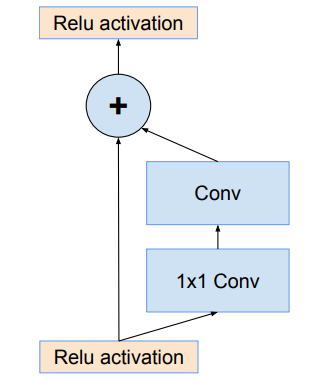
1. Describe the Inception block.

Inception V4 was introduced in combination with Inception-ResNet by the researchers a Google in 2016. The main aim of the paper was to reduce the complexity of Inception V3 model which give the state-of-the-art accuracy on ILSVRC 2015 challenge. This paper also explores the possibility of using residual networks on Inception model. This model

**Architectural Changes in Inception-V4:**

 In the paper there are two types of Inception architectures were discussed.

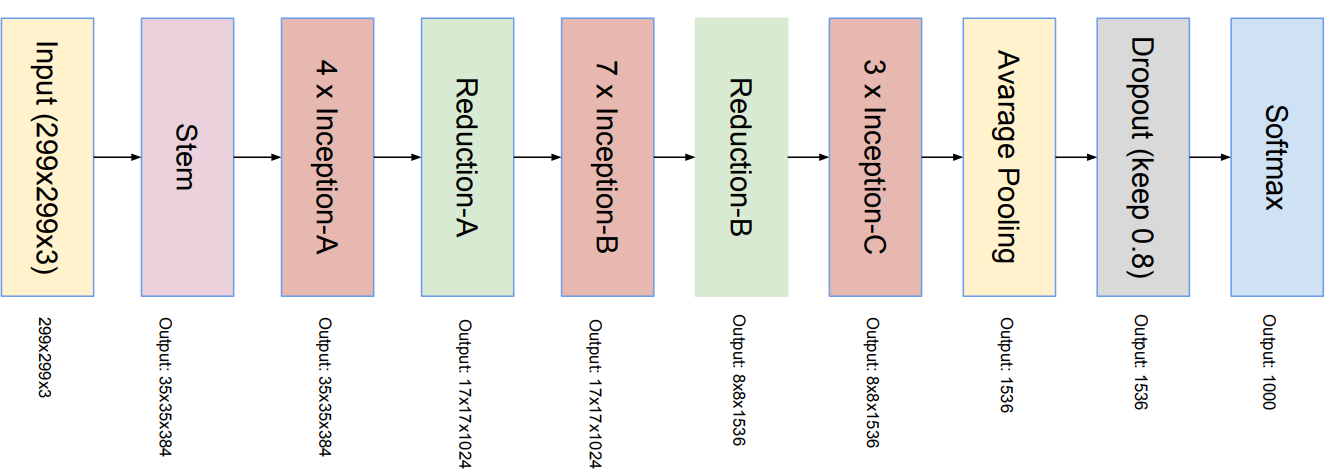
* **Pure Inception architecture (Inception -V4):**
  + The initial set of layers which the paper refers “stem of the architecture” was modified to make it more uniform . These layers are used before Inception block in the architecture.
  + This model can be trained without partition of replicas unlike the previous versions of inceptions which required different replica in order to fit in memory. This architecture use memory optimization on back propagation to reduce the memory requirement.
* **Inception architecture with residuals:**
  + The authors of the paper was inspired by the success of Residual Network. Therefore they explored the possibility of combining the Inception with ResNets. They proposed two Residual Network based Inception models: Inception ResNet V1 and Inception ResNet V2. Let’s look at the key highlights of these architectures.

[](https://media.geeksforgeeks.org/wp-content/uploads/20200502220103/Inception-ResNet-connection.PNG)

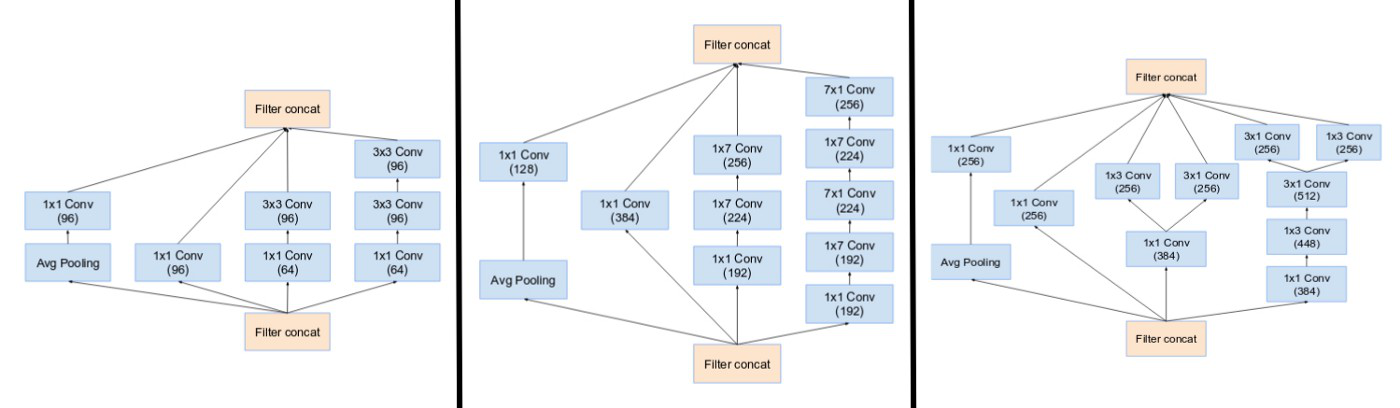
* The Inception block used in these architecture are computationally less expensive than original Inception blocks that we used in Inception V4.
* Each Inception block is followed by a 1×1 convolution without activation called filter expansion. This is done to scale up the dimensionality of filter bank to match the depth of input to next layer.
* The pooling operation inside the Inception blocks were replaced by residual connections. However, pooling operations can be found in reduction blocks.
* In Inception ResNets models, the batch normalization does not used after summations. This is done to reduce the model size to make it trainable on a single GPU.
* Both the Inception architectures have same architectures for Reduction Blocks, but have different stem of the architectures. They also have difference in their hyper parameters for training.
* It is found that Inception-ResNet V1 have similar computational cost as of Inception V3 and Inception-ResNet V2 have similar computational cost as of Inception V4.

**Architectures:**

* Below is the architectural details of **Inception V4:**
  + Overall Architecture

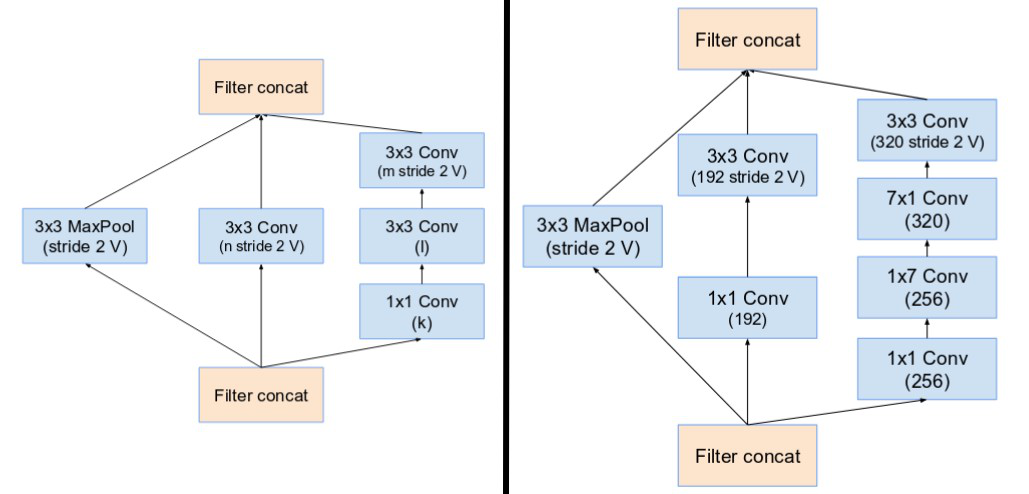


* Inception modules:



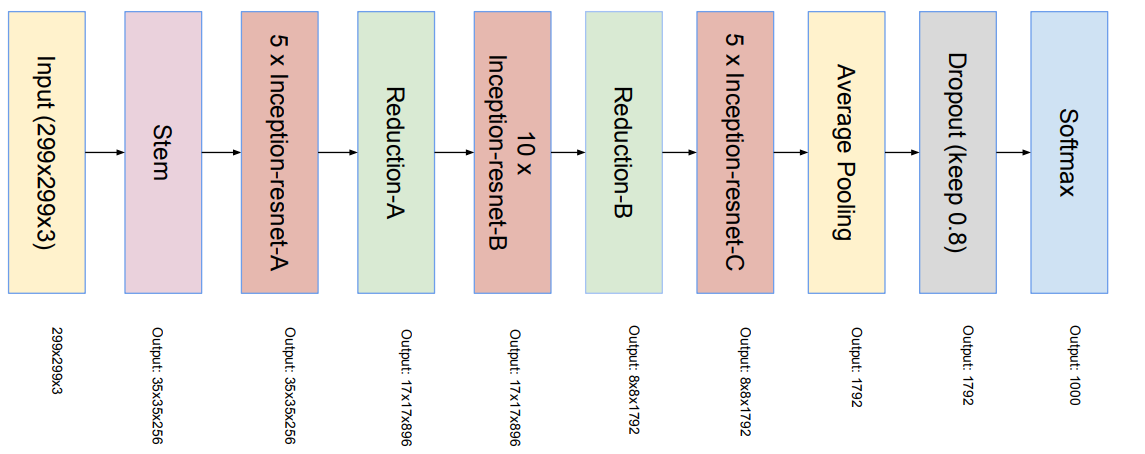
*Inception Modules A, B, C of Inception-v4*

* Reduction Modules:



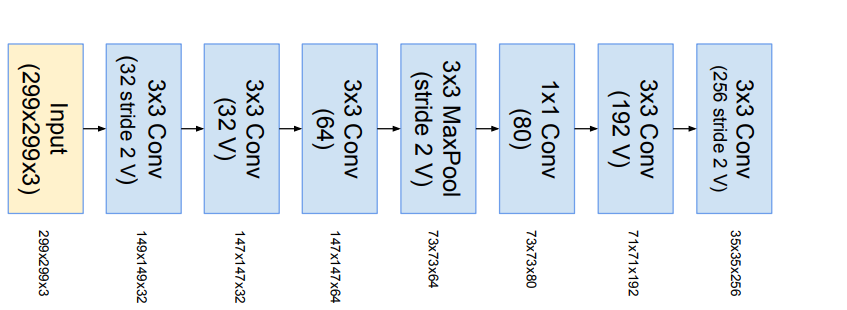
*Reduction Blocks A, B of Inception-v4*

* Below is the architectural details of **Inception ResNet V1 and Inception** **ResNet V2**:
  + Overall Architectures: Inception ResNet V2 has similar architecture schema as of V1 but the difference lies in their stems,  Inception and Reduction blocks.

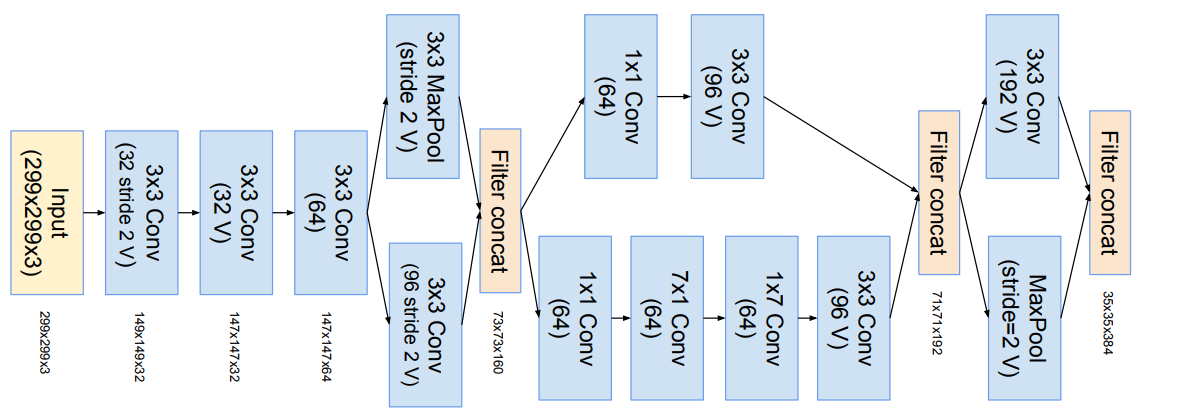
[](https://media.geeksforgeeks.org/wp-content/uploads/20200502221704/Inception-ResNet.PNG)

*Inception ResNet V1 and Inception ResNet V2*

* Stem of the architecture

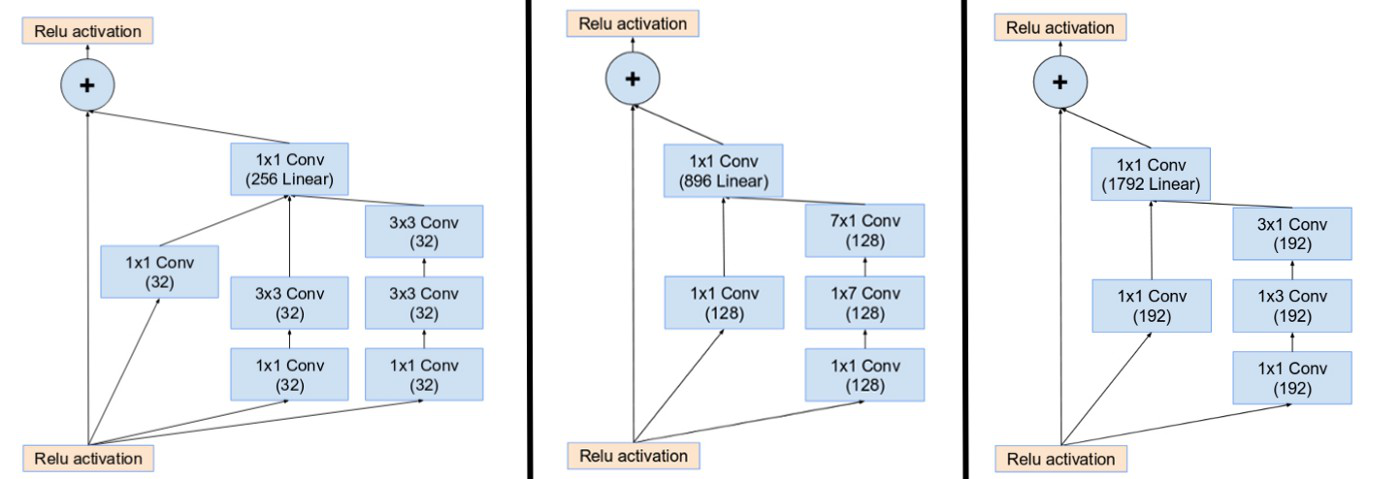


*Inception ResNet v1 stem*

[](https://media.geeksforgeeks.org/wp-content/uploads/20200502221129/stem-InceptionV4-and-Inception-ResNet-V2.PNG)

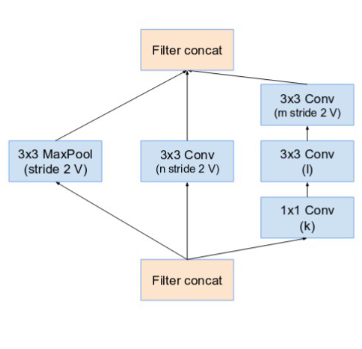
*Inception ResNet V2 stem*

* Inception Blocks: Inception blocks in Inception ResNets are very similar except for few changes in number of parameters. In Inception ResNet V2 the number of parameters increase in some layers in comparison to Inception ResNet V1.



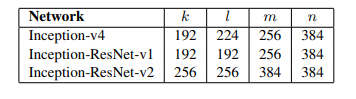
*Inception modules A, B, C of Inception ResNet V1*

* Reduction Blocks:



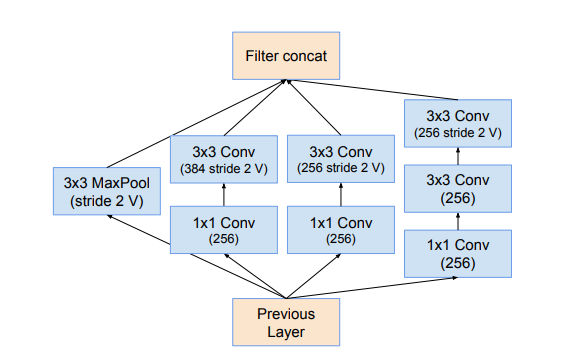
*Reduction A schema*

* The reduction module A in different Inception architecture is similar. The only difference in number of parameters that are defined by table below:

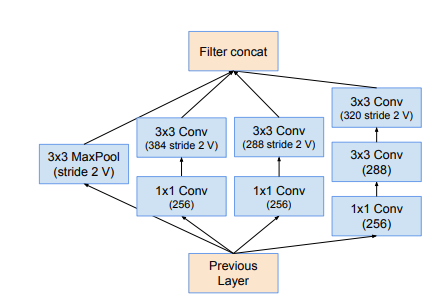


*Hyper parameters of Inception-v4*

* The Reduction Block B for Inception ResNets are given below:



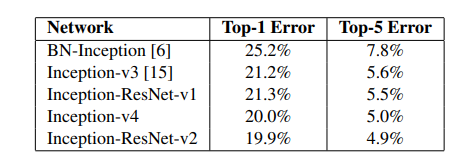
*Inception ResNet-v1 Reduction Block B*



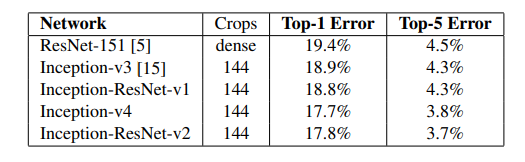
*Inception ResNet-v2 Reduction Block B*

**Results and Conclusion:**

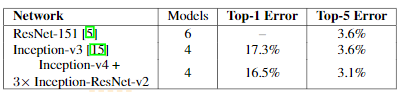
The top-5 and top-1 error rate of single-crop single-model evaluation of different architectures on the ILSVRC 2012 validation sets are below:



The top-5 and top-1 error rate of 144-crop (single-model) evaluation of different architectures on the ILSVRC 2012 validation sets are below:



The result on ensemble of different architectures on the ILSVRC 2012 validation sets are below:

[](https://media.geeksforgeeks.org/wp-content/uploads/20200502221850/Inception-V4-ensemble.PNG)

1. What is the DIMENSIONALITY REDUCTION LAYER (1 LAYER CONVOLUTIONAL)?

When we do a standard convolution of say a 6x6 image with three color channels(*depth -3*) , Fig 1. below, with a single filter of dimension 3x3x3 (*depth of filter has to match input volume depth*) we get as output 4x4x1 (*assuming stride 1 and no padding*). The key point to note here is the output is collapsed from depth 3 to depth 1 *(granted width and height changed too but we could have kept that same as input by proper choice of padding of input)*.

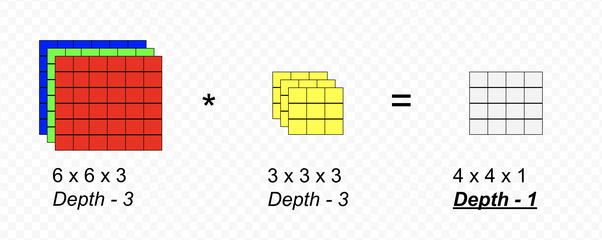


Fig 1.

Now instead of using a 3 x 3 x 3 filter, if we use a 1 x 1 x 3*(often called 1x1 since the depth is a variable and forced to match input volume depth - which is perhaps why it is so confusing), fig 2.*the output again has depth 1 as in previous case (fig 1), except since we convolved with a 1x1 filter, the width and height of the input remains unchanged.

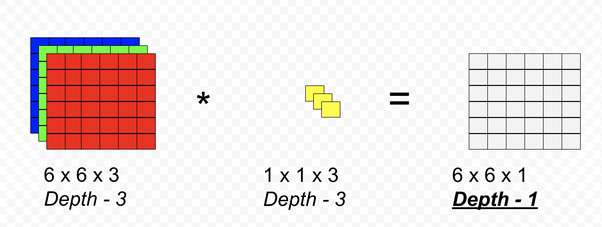


Fig 2.

However, if we increase the number of filters we can control the depth of the output. For instance, using two filters *(each of depth 3)* in the figure below , fig 3, the output depth is 2.

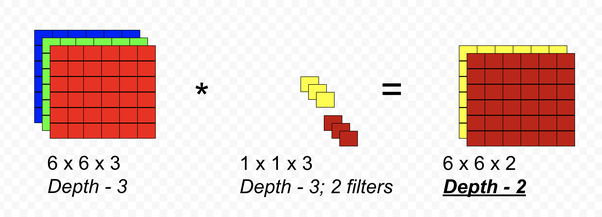


Fig 3.

Finally in the case of Fig 4 below, where we use the 3 filters, the input and output volumes are exactly the same dimension. So the filters serve as a means to add a nonlinearity to the input volume *(though I don’t know/think if it is ever used this way)*

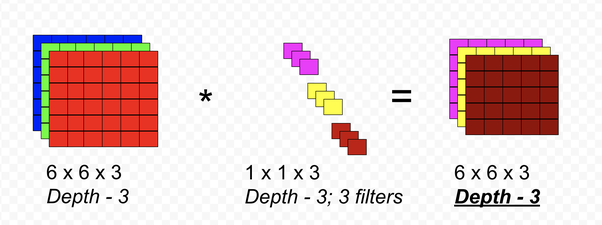
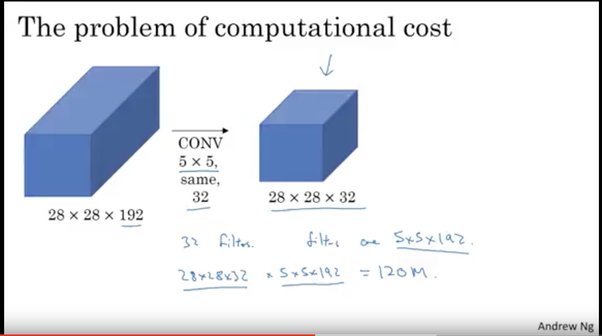


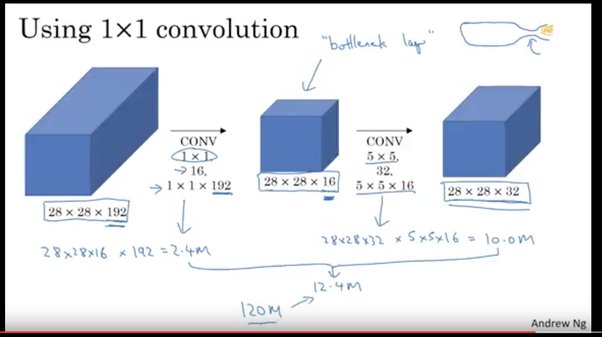
Fig 4.

In summary, 1x1 convolutions serve as a means to

* control the depth of the input volume as it is passed to the next layer, either decrease it, or increase it, or just add a non-linearity when it doesn’t alter the depth. This control is achieved by the choosing the appropriate number of filters. We can control the other two dimensions - width and height by the filter sizes and padding parameters, or use pooling to reduce width and height.
* In the case when it is reduces the dimensions, it is a means to reduce computations - can be an order of magnitude less as shown in the example from Andrew’s lectures fig. 5 (without 1x1) and fig 6 (with 1x1).



*Fig 5. without 1x1 - 120 million computations. The word “same” above means a padding ( in this case it is 2) is used to keep output volume same as input volume. Andrew explains “valid” ( no padding ) and “same” ( with padding) in his video titled “padding”*



*Fig 6. with 1x1 - 12.4 computations*

1. THE IMPACT OF REDUCING DIMENSIONALITY ON NETWORK PERFORMANCE

Introduction to Dimensionality Reduction Technique

What is Dimensionality Reduction?

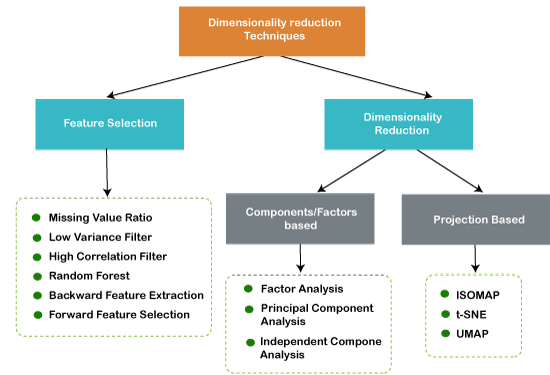
The number of input features, variables, or columns present in a given dataset is known as dimensionality, and the process to reduce these features is called dimensionality reduction.

A dataset contains a huge number of input features in various cases, which makes the predictive modeling task more complicated. Because it is very difficult to visualize or make predictions for the training dataset with a high number of features, for such cases, dimensionality reduction techniques are required to use.

Dimensionality reduction technique can be defined as, ***"It is a way of converting the higher dimensions dataset into lesser dimensions dataset ensuring that it provides similar information."*** These techniques are widely used in [machine learning](https://www.javatpoint.com/machine-learning) for obtaining a better fit predictive model while solving the classification and regression problems.

It is commonly used in the fields that deal with high-dimensional data, such as **speech recognition, signal processing, bioinformatics, etc. It can also be used for data visualization, noise reduction, cluster analysis**, etc.

Backward Skip 10sPlay VideoForward Skip 10s



The Curse of Dimensionality

Handling the high-dimensional data is very difficult in practice, commonly known as the *curse of dimensionality.* If the dimensionality of the input dataset increases, any machine learning algorithm and model becomes more complex. As the number of features increases, the number of samples also gets increased proportionally, and the chance of overfitting also increases. If the machine learning model is trained on high-dimensional data, it becomes overfitted and results in poor performance.

Hence, it is often required to reduce the number of features, which can be done with dimensionality reduction.

Benefits of applying Dimensionality Reduction

Some benefits of applying dimensionality reduction technique to the given dataset are given below:

* By reducing the dimensions of the features, the space required to store the dataset also gets reduced.
* Less Computation training time is required for reduced dimensions of features.
* Reduced dimensions of features of the dataset help in visualizing the data quickly.
* **I**t removes the redundant features (if present) by taking care of multicollinearity.

Disadvantages of dimensionality Reduction

There are also some disadvantages of applying the dimensionality reduction, which are given below:

* Some data may be lost due to dimensionality reduction.
* In the PCA dimensionality reduction technique, sometimes the principal components required to consider are unknown.

Approaches of Dimension Reduction

There are two ways to apply the dimension reduction technique, which are given below:

Feature Selection

Feature selection is the process of selecting the subset of the relevant features and leaving out the irrelevant features present in a dataset to build a model of high accuracy. In other words, it is a way of selecting the optimal features from the input dataset.

Three methods are used for the feature selection:

**1. Filters Methods**

In this method, the dataset is filtered, and a subset that contains only the relevant features is taken. Some common techniques of filters method are:

* **Correlation**
* **Chi-Square Test**
* **ANOVA**
* **Information Gain, etc.**

**2. Wrappers Methods**

The wrapper method has the same goal as the filter method, but it takes a machine learning model for its evaluation. In this method, some features are fed to the ML model, and evaluate the performance. The performance decides whether to add those features or remove to increase the accuracy of the model. This method is more accurate than the filtering method but complex to work. Some common techniques of wrapper methods are:

* Forward Selection
* Backward Selection
* Bi-directional Elimination

**3. Embedded Methods:** Embedded methods check the different training iterations of the machine learning model and evaluate the importance of each feature. Some common techniques of Embedded methods are:

* **LASSO**
* **Elastic Net**
* **Ridge Regression, etc.**

Feature Extraction:

Feature extraction is the process of transforming the space containing many dimensions into space with fewer dimensions. This approach is useful when we want to keep the whole information but use fewer resources while processing the information.

Some common feature extraction techniques are:

* Principal Component Analysis
* Linear Discriminant Analysis
* Kernel PCA
* Quadratic Discriminant Analysis

Common techniques of Dimensionality Reduction

1. **Principal Component Analysis**
2. **Backward Elimination**
3. **Forward Selection**
4. **Score comparison**
5. **Missing Value Ratio**
6. **Low Variance Filter**
7. **High Correlation Filter**
8. **Random Forest**
9. **Factor Analysis**
10. **Auto-Encoder**

Principal Component Analysis (PCA)

Principal Component Analysis is a statistical process that converts the observations of correlated features into a set of linearly uncorrelated features with the help of orthogonal transformation. These new transformed features are called the **Principal Components**. It is one of the popular tools that is used for exploratory data analysis and predictive modeling.

PCA works by considering the variance of each attribute because the high attribute shows the good split between the classes, and hence it reduces the dimensionality. Some real-world applications of PCA are ***image processing, movie recommendation system, optimizing the power allocation in various communication channels.***

Backward Feature Elimination

The backward feature elimination technique is mainly used while developing Linear Regression or Logistic Regression model. Below steps are performed in this technique to reduce the dimensionality or in feature selection:

* In this technique, firstly, all the n variables of the given dataset are taken to train the model.
* The performance of the model is checked.
* Now we will remove one feature each time and train the model on n-1 features for n times, and will compute the performance of the model.
* We will check the variable that has made the smallest or no change in the performance of the model, and then we will drop that variable or features; after that, we will be left with n-1 features.
* Repeat the complete process until no feature can be dropped.

In this technique, by selecting the optimum performance of the model and maximum tolerable error rate, we can define the optimal number of features require for the machine learning algorithms.

Forward Feature Selection

Forward feature selection follows the inverse process of the backward elimination process. It means, in this technique, we don't eliminate the feature; instead, we will find the best features that can produce the highest increase in the performance of the model. Below steps are performed in this technique:

* We start with a single feature only, and progressively we will add each feature at a time.
* Here we will train the model on each feature separately.
* The feature with the best performance is selected.
* The process will be repeated until we get a significant increase in the performance of the model.

Missing Value Ratio

If a dataset has too many missing values, then we drop those variables as they do not carry much useful information. To perform this, we can set a threshold level, and if a variable has missing values more than that threshold, we will drop that variable. The higher the threshold value, the more efficient the reduction.

Low Variance Filter

As same as missing value ratio technique, data columns with some changes in the data have less information. Therefore, we need to calculate the variance of each variable, and all data columns with variance lower than a given threshold are dropped because low variance features will not affect the target variable.

High Correlation Filter

High Correlation refers to the case when two variables carry approximately similar information. Due to this factor, the performance of the model can be degraded. This correlation between the independent numerical variable gives the calculated value of the correlation coefficient. If this value is higher than the threshold value, we can remove one of the variables from the dataset. We can consider those variables or features that show a high correlation with the target variable.

Random Forest

Random Forest is a popular and very useful feature selection algorithm in machine learning. This algorithm contains an in-built feature importance package, so we do not need to program it separately. In this technique, we need to generate a large set of trees against the target variable, and with the help of usage statistics of each attribute, we need to find the subset of features.

Random forest algorithm takes only numerical variables, so we need to convert the input data into numeric data using **hot encoding**.

Factor Analysis

Factor analysis is a technique in which each variable is kept within a group according to the correlation with other variables, it means variables within a group can have a high correlation between themselves, but they have a low correlation with variables of other groups.

We can understand it by an example, such as if we have two variables Income and spend. These two variables have a high correlation, which means people with high income spends more, and vice versa. So, such variables are put into a group, and that group is known as the **factor**. The number of these factors will be reduced as compared to the original dimension of the dataset.

Auto-encoders

One of the popular methods of dimensionality reduction is auto-encoder, which is a type of ANN or [artificial neural network](https://www.javatpoint.com/artificial-neural-network), and its main aim is to copy the inputs to their outputs. In this, the input is compressed into latent-space representation, and output is occurred using this representation. It has mainly two parts:

* **Encoder:** The function of the encoder is to compress the input to form the latent-space representation.
* **Decoder:** The function of the decoder is to recreate the output from the latent-space representation.

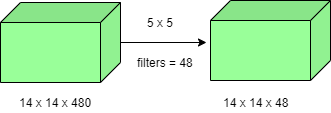
1. Mention three components. Style GoogLeNet

Google Net (or Inception V1) was proposed by research at Google (with the collaboration of various universities) in 2014 in the research paper titled “Going Deeper with Convolutions”. This architecture was the winner at the ILSVRC 2014 image classification challenge. It has provided a significant decrease in error rate as compared to previous winners AlexNet (Winner of ILSVRC 2012) and ZF-Net (Winner of ILSVRC 2013) and significantly less error rate than VGG (2014 runner up). This architecture uses techniques such as *1×1* convolutions in the middle of the architecture and global average pooling.

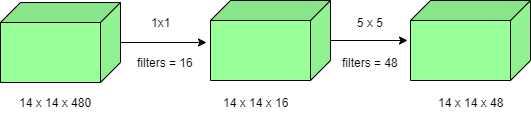
**Features of GoogleNet:**

The GoogLeNet architecture is very different from previous state-of-the-art architectures such as AlexNet and ZF-Net. It uses many different kinds of methods such as *1×1* convolution and global average pooling that enables it to create deeper architecture. In the architecture, we will discuss some of these methods:

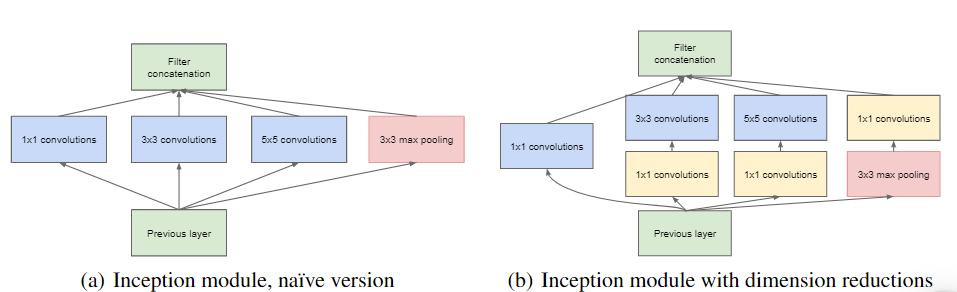
* ***1×1* convolution :**The inception architecture uses *1×1* convolution in its architecture. These convolutions used to decrease the number of parameters (weights and biases) of the architecture. By reducing the parameters we also increase the depth of the architecture. Let’s look at an example of a *1×1* convolution below:
  + For Example, If we want to perform *5×5* convolution having 48 filters without using *1×1* convolution as intermediate:

[](https://media.geeksforgeeks.org/wp-content/uploads/20200429201100/without1x1.png)

* Total Number of operations : *(14 x 14 x 48) x (5 x 5 x 480) = 112.9 M*
  + With 1×1 convolution :

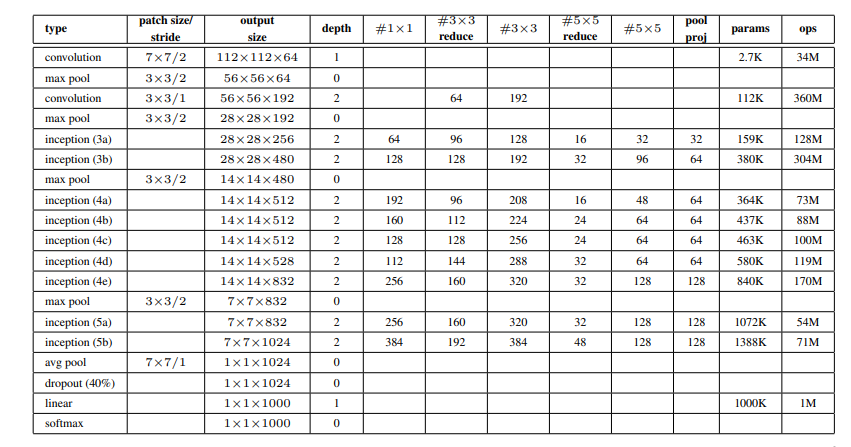
[](https://media.geeksforgeeks.org/wp-content/uploads/20200429201229/with1x1.png)

* *(14 x 14 x 16) x (1 x 1 x 480) + (14 x 14 x 48) x (5 x 5 x 16) = 1.5M + 3.8M = 5.3M* which is much smaller than 112.9M.
* **Global Average Pooling :**  
  In the previous architecture such as AlexNet, the fully connected layers are used at the end of the network. These fully connected layers contain the majority of parameters of many architectures that causes an increase in computation cost.  
  In GoogLeNet architecture, there is a method called global average pooling is used at the end of the network. This layer takes a feature map of *7×7* and averages it to *1×1*. This also decreases the number of trainable parameters to 0 and improves the top-1 accuracy by 0.6%
* **Inception Module:**  
  The inception module is different from previous architectures such as AlexNet, ZF-Net. In this architecture, there is a fixed convolution size for each layer.  
  In the Inception module *1×1, 3×3, 5×5* convolution and *3×3* max pooling performed in a parallel way at the input and the output of these are stacked together to generated final output. The idea behind that convolution filters of different sizes will handle objects at multiple scale better.

[](https://media.geeksforgeeks.org/wp-content/uploads/20200429201304/Incepption-module.PNG)

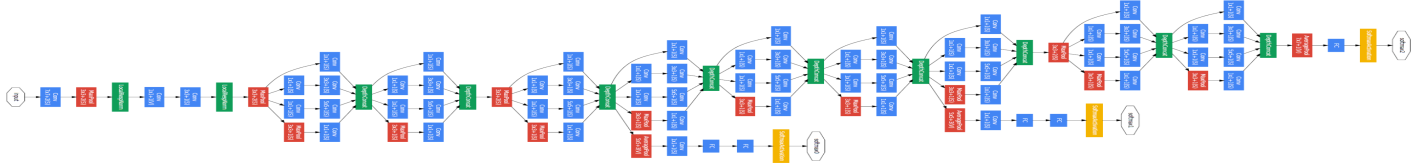
* **Auxiliary Classifier for Training:**  
  Inception architecture used some intermediate classifier branches in the middle of the architecture, these branches are used during training only. These branches consist of a 5×5 average pooling layer with a stride of 3, a *1×1* convolutions with *128* filters, two fully connected layers of 1024 outputs and 1000 outputs and a softmax classification layer. The generated loss of these layers added to total loss with a weight of 0.3. These layers help in combating gradient vanishing problem and also provide regularization.

**Model Architecture:**

Below is Layer by Layer architectural details of GoogLeNet.  
[](https://media.geeksforgeeks.org/wp-content/uploads/20200429201421/Inception-layer-by-layer.PNG)  
The overall architecture is 22 layers deep. The architecture was designed to keep computational efficiency in mind. The idea behind that the architecture can be run on individual devices even with low computational resources. The architecture also contains two auxiliary classifier layer connected to the output of Inception (4a) and Inception (4d) layers.

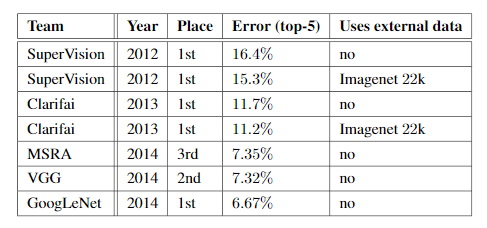
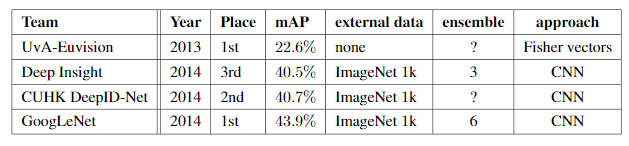
The architectural details of auxiliary classifiers as follows:

* An average pooling layer of filter size 5×5 and stride 3.
* A 1×1 convolution with 128 filters for dimension reduction and ReLU activation.
* A fully connected layer with 1025 outputs and ReLU activation
* Dropout Regularization with dropout ratio = 0.7
* A softmax classifier with 1000 classes output similar to the main softmax classifier.

[](https://media.geeksforgeeks.org/wp-content/uploads/20200429201549/Inceptionv1_architecture.png)  
This architecture takes image of size *224 x 224* with RGB color channels. All the convolutions inside this architecture uses Rectified Linear Units (ReLU) as their activation functions.

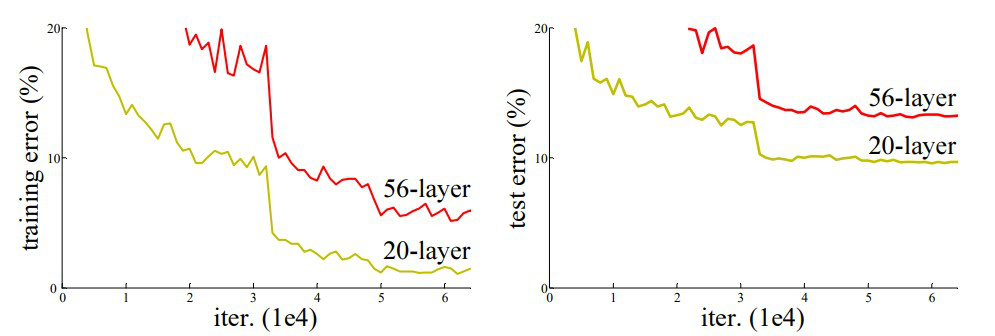
**Results:**

GoogLeNet was the winner at ILSRVRC 2014 taking 1st place in both classification an detection task. It has top-5 error rate of 6.67% in classification task. An ensemble of 6 GoogLeNets gives 43.9 % mAP on ImageNet test set.

[](https://media.geeksforgeeks.org/wp-content/uploads/20200429201653/GoogleNet-classification-performance.PNG)[](https://media.geeksforgeeks.org/wp-content/uploads/20200429201717/GoogleNet-Detection-Performance.PNG)

1. Using our own terms and diagrams, explain RESNET ARCHITECTURE.

After the first CNN-based architecture (AlexNet) that win the ImageNet 2012 competition, Every subsequent winning architecture uses more layers in a deep neural network to reduce the error rate. This works for less number of layers, but when we increase the number of layers, there is a common problem in deep learning associated with that called the Vanishing/Exploding gradient. This causes the gradient to become 0 or too large. Thus when we increases number of layers, the training and test error rate also increases.

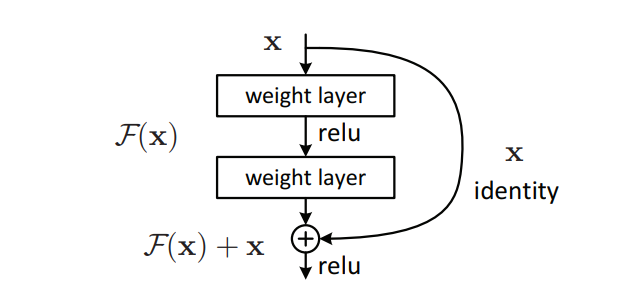
[](https://media.geeksforgeeks.org/wp-content/uploads/20200424200128/abc.jpg)

*Comparison of 20-layer vs 56-layer architecture*

In the above plot, we can observe that a 56-layer CNN gives more error rate on both training and testing dataset than a 20-layer CNN architecture. After analyzing more on error rate the authors were able to reach conclusion that it is caused by vanishing/exploding gradient.   
ResNet, which was proposed in 2015 by researchers at Microsoft Research introduced a new architecture called Residual Network.

**Residual Network:**In order to solve the problem of the vanishing/exploding gradient, this architecture introduced the concept called Residual Blocks. In this network, we use a technique called ***skip connections***. The skip connection connects activations of a  layer to further layers by skipping some layers in between. This forms a residual block. Resnets are made by stacking these residual blocks together.   
The approach behind this network is instead of layers learning the underlying mapping, we allow the network to fit the residual mapping. So, instead of say H(x), initial mapping*,*let the network fit,

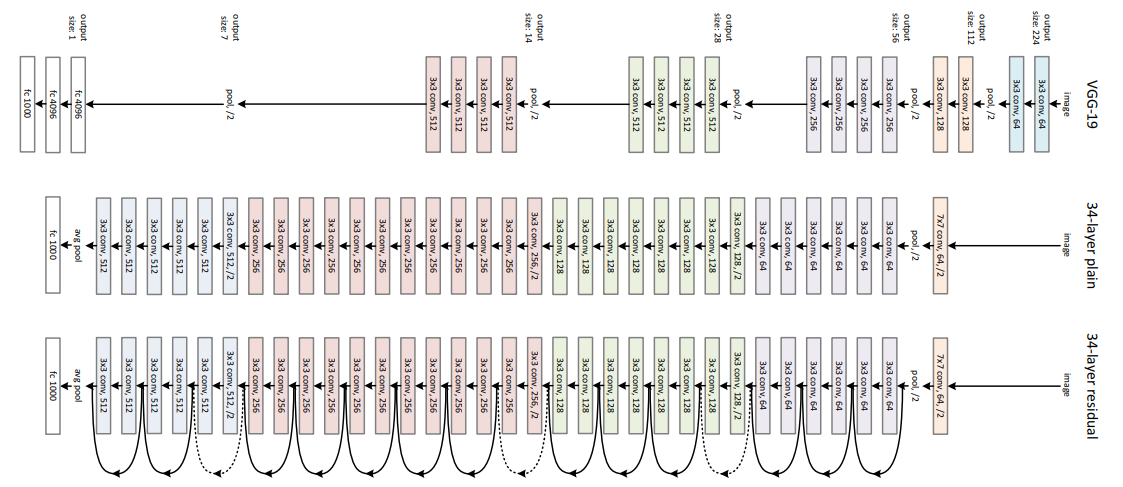
*F(x) := H(x) - x* which gives *H(x) := F(x) + x*.

[](https://media.geeksforgeeks.org/wp-content/uploads/20200424011510/Residual-Block.PNG)

*Skip (Shortcut) connection*

The advantage of adding this type of skip connection is that if any layer hurt the performance of architecture then it will be skipped by regularization. So, this results in training a very deep neural network without the problems caused by vanishing/exploding gradient.  The authors of the paper experimented on 100-1000 layers of the CIFAR-10 dataset.   
There is a similar approach called “highway networks”, these networks also use skip connection. Similar to LSTM these skip connections also use parametric gates. These gates determine how much information passes through the skip connection. This architecture however has not provided accuracy better than ResNet architecture.

**Network Architecture:** This network uses a 34-layer plain network architecture inspired by VGG-19 in which then the shortcut connection is added. These shortcut connections then convert the architecture into a residual network.  

[](https://media.geeksforgeeks.org/wp-content/uploads/20200424011138/ResNet.PNG)

*ResNet -34 architecture*

**Implementation:** Using the Tensorflow and Keras API, we can design ResNet architecture (including Residual Blocks) from scratch. Below is the implementation of different ResNet architecture. For this implementation, we use the CIFAR-10 dataset. This dataset contains 60, 000 32×32 color images in 10 different classes (airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks), etc. This dataset can be assessed from k*eras.datasets* API function.

**Step 1:** First, we import the keras module and its APIs. These APIs help in building the architecture of the ResNet model.

**Code:**Importing libraries

# Import Keras modules and its important APIs

import keras

from keras.layers import Dense, Conv2D, BatchNormalization, Activation

from keras.layers import AveragePooling2D, Input, Flatten

from keras.optimizers import Adam

from keras.callbacks import ModelCheckpoint, LearningRateScheduler

from keras.callbacks import ReduceLROnPlateau

from keras.preprocessing.image import ImageDataGenerator

from keras.regularizers import l2

from keras import backend as K

from keras.models import Model

from keras.datasets import cifar10

import numpy as np

import os

**Step 2:** Now, We set different hyper parameters that are required for ResNet architecture. We also did some preprocessing on our dataset to prepare it for training.

**Code:**Setting Training Hyperparameters

* python3

|  |
| --- |
| # Setting Training Hyperparameters  batch\_size = 32  # original ResNet paper uses batch\_size = 128 for training  epochs = 200  data\_augmentation = True  num\_classes = 10    # Data Preprocessing  subtract\_pixel\_mean = True  n = 3    # Select ResNet Version  version = 1    # Computed depth of  if version == 1:      depth = n \* 6 + 2  elif version == 2:      depth = n \* 9 + 2    # Model name, depth and version  model\_type = 'ResNet % dv % d' % (depth, version)    # Load the CIFAR-10 data.  (x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()    # Input image dimensions.  input\_shape = x\_train.shape[1:]    # Normalize data.  x\_train = x\_train.astype('float32') / 255  x\_test = x\_test.astype('float32') / 255    # If subtract pixel mean is enabled  if subtract\_pixel\_mean:      x\_train\_mean = np.mean(x\_train, axis = 0)      x\_train -= x\_train\_mean      x\_test -= x\_train\_mean    # Print Training and Test Samples  print('x\_train shape:', x\_train.shape)  print(x\_train.shape[0], 'train samples')  print(x\_test.shape[0], 'test samples')  print('y\_train shape:', y\_train.shape)    # Convert class vectors to binary class matrices.  y\_train = keras.utils.to\_categorical(y\_train, num\_classes)  y\_test = keras.utils.to\_categorical(y\_test, num\_classes) |

**Step 3:** In this step, we set the learning rate according to the number of epochs. As the number of epochs the learning rate must be decreased to ensure better learning.

**Code:**Setting LR for different numbers of Epochs

* python3

|  |
| --- |
| # Setting LR for different number of Epochs  def lr\_schedule(epoch):      lr = 1e-3      if epoch > 180:          lr \*= 0.5e-3      elif epoch > 160:          lr \*= 1e-3      elif epoch > 120:          lr \*= 1e-2      elif epoch > 80:          lr \*= 1e-1      print('Learning rate: ', lr)      return lr |

**Step 4:** Define basic ResNet building block that can be used for defining the ResNet V1 and V2 architecture.

**Code:**Basic ResNet Building Block

* python3

|  |
| --- |
| # Basic ResNet Building Block      def resnet\_layer(inputs,                   num\_filters=16,                   kernel\_size=3,                   strides=1,                   activation='relu',                   batch\_normalization=True,      conv=Conv2D(num\_filters,                    kernel\_size=kernel\_size,                    strides=strides,                    padding='same',                    kernel\_initializer='he\_normal',                    kernel\_regularizer=l2(1e-4))        x=inputs      if conv\_first:          x = conv(x)          if batch\_normalization:              x = BatchNormalization()(x)          if activation is not None:              x = Activation(activation)(x)      else:          if batch\_normalization:              x = BatchNormalization()(x)          if activation is not None:              x = Activation(activation)(x)          x = conv(x)      return x |

**Step 5:** Define ResNet V1 architecture that is based on the ResNet building block we defined above:

**Code:**ResNet V1 architecture

* python3

|  |
| --- |
| def resnet\_v1(input\_shape, depth, num\_classes=10):        if (depth - 2) % 6 != 0:          raise ValueError('depth should be 6n + 2 (eg 20, 32, 44 in [a])')      # Start model definition.      num\_filters = 16      num\_res\_blocks = int((depth - 2) / 6)        inputs = Input(shape=input\_shape)      x = resnet\_layer(inputs=inputs)      # Instantiate the stack of residual units      for stack in range(3):          for res\_block in range(num\_res\_blocks):              strides = 1              if stack & gt              0 and res\_block == 0:  # first layer but not first stack                  strides = 2  # downsample              y = resnet\_layer(inputs=x,                               num\_filters=num\_filters,                               strides=strides)              y = resnet\_layer(inputs=y,                               num\_filters=num\_filters,                               activation=None)              if stack & gt              0 and res\_block == 0:  # first layer but not first stack                  # linear projection residual shortcut connection to match                  # changed dims                  x = resnet\_layer(inputs=x,                                   num\_filters=num\_filters,                                   kernel\_size=1,                                   strides=strides,                                   activation=None,                                   batch\_normalization=False)              x = keras.layers.add([x, y])              x = Activation('relu')(x)          num\_filters \*= 2        # Add classifier on top.      # v1 does not use BN after last shortcut connection-ReLU      x = AveragePooling2D(pool\_size=8)(x)      y = Flatten()(x)      outputs = Dense(num\_classes,                      activation='softmax',                      kernel\_initializer='he\_normal')(y)        # Instantiate model.      model = Model(inputs=inputs, outputs=outputs)      return model |

**Step 6:** Define ResNet V2 architecture that is based on the ResNet building block we defined above:

**Code:**ResNet V2 architecture

* python3

|  |
| --- |
| # ResNet V2 architecture  def resnet\_v2(input\_shape, depth, num\_classes=10):      if (depth - 2) % 9 != 0:          raise ValueError('depth should be 9n + 2 (eg 56 or 110 in [b])')      # Start model definition.      num\_filters\_in = 16      num\_res\_blocks = int((depth - 2) / 9)        inputs = Input(shape=input\_shape)      # v2 performs Conv2D with BN-ReLU on input before splitting into 2 paths      x = resnet\_layer(inputs=inputs,                       num\_filters=num\_filters\_in,                       conv\_first=True)        # Instantiate the stack of residual units      for stage in range(3):          for res\_block in range(num\_res\_blocks):              activation = 'relu'              batch\_normalization = True              strides = 1              if stage == 0:                  num\_filters\_out = num\_filters\_in \* 4                  if res\_block == 0:  # first layer and first stage                      activation = None                      batch\_normalization = False              else:                  num\_filters\_out = num\_filters\_in \* 2                  if res\_block == 0:  # first layer but not first stage                      strides = 2    # downsample                # bottleneck residual unit              y = resnet\_layer(inputs=x,                               num\_filters=num\_filters\_in,                               kernel\_size=1,                               strides=strides,                               activation=activation,                               batch\_normalization=batch\_normalization,                               conv\_first=False)              y = resnet\_layer(inputs=y,                               num\_filters=num\_filters\_in,                               conv\_first=False)              y = resnet\_layer(inputs=y,                               num\_filters=num\_filters\_out,                               kernel\_size=1,                               conv\_first=False)              if res\_block == 0:                  # linear projection residual shortcut connection to match                  # changed dims                  x = resnet\_layer(inputs=x,                                   num\_filters=num\_filters\_out,                                   kernel\_size=1,                                   strides=strides,                                   activation=None,                                   batch\_normalization=False)              x = keras.layers.add([x, y])            num\_filters\_in = num\_filters\_out        # Add classifier on top.      # v2 has BN-ReLU before Pooling      x = BatchNormalization()(x)      x = Activation('relu')(x)      x = AveragePooling2D(pool\_size=8)(x)      y = Flatten()(x)      outputs = Dense(num\_classes,                      activation='softmax',                      kernel\_initializer='he\_normal')(y)        # Instantiate model.      model = Model(inputs=inputs, outputs=outputs)      return model |

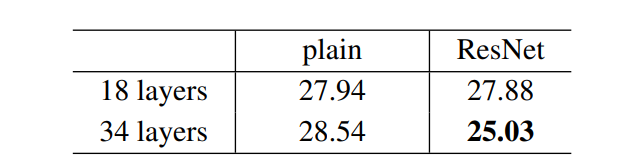
**Step 7:** The code below is used to train and test the ResNet v1 and v2 architecture we defined above:

**Code: Main function**

* python3

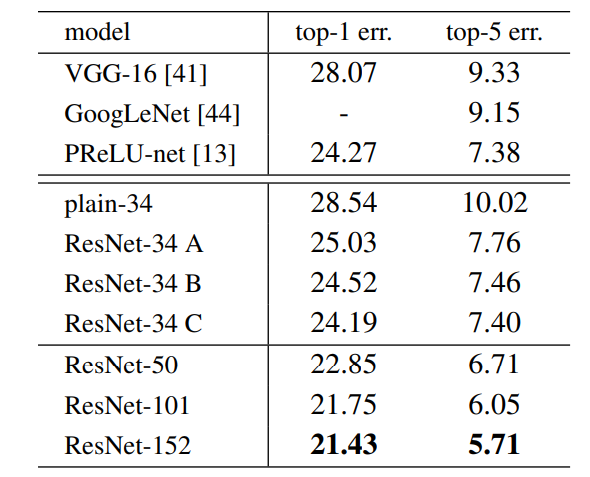
|  |
| --- |
| # Main function  if version == 2:      model = resnet\_v2(input\_shape = input\_shape, depth = depth)  else:      model = resnet\_v1(input\_shape = input\_shape, depth = depth)    model.compile(loss ='categorical\_crossentropy',                optimizer = Adam(learning\_rate = lr\_schedule(0)),                metrics =['accuracy'])  model.summary()  print(model\_type)    # Prepare model saving directory.  save\_dir = os.path.join(os.getcwd(), 'saved\_models')  model\_name = 'cifar10\_% s\_model.{epoch:03d}.h5' % model\_type  if not os.path.isdir(save\_dir):      os.makedirs(save\_dir)  filepath = os.path.join(save\_dir, model\_name)    # Prepare callbacks for model saving and for learning rate adjustment.  checkpoint = ModelCheckpoint(filepath = filepath,                               monitor ='val\_acc',                               verbose = 1,                               save\_best\_only = True)    lr\_scheduler = LearningRateScheduler(lr\_schedule)    lr\_reducer = ReduceLROnPlateau(factor = np.sqrt(0.1),                                 cooldown = 0,                                 patience = 5,                                 min\_lr = 0.5e-6)    callbacks = [checkpoint, lr\_reducer, lr\_scheduler]    # Run training, with or without data augmentation.  if not data\_augmentation:      print('Not using data augmentation.')      model.fit(x\_train, y\_train,                batch\_size = batch\_size,                epochs = epochs,                validation\_data =(x\_test, y\_test),                shuffle = True,                callbacks = callbacks)  else:      print('Using real-time data augmentation.')      # This will do preprocessing and realtime data augmentation:      datagen = ImageDataGenerator(          # set input mean to 0 over the dataset          featurewise\_center = False,          # set each sample mean to 0          samplewise\_center = False,          # divide inputs by std of dataset          featurewise\_std\_normalization = False,          # divide each input by its std          samplewise\_std\_normalization = False,          # apply ZCA whitening          zca\_whitening = False,          # epsilon for ZCA whitening          zca\_epsilon = 1e-06,          # randomly rotate images in the range (deg 0 to 180)          rotation\_range = 0,          # randomly shift images horizontally          width\_shift\_range = 0.1,          # randomly shift images vertically          height\_shift\_range = 0.1,          # set range for random shear          shear\_range = 0.,          # set range for random zoom          zoom\_range = 0.,          # set range for random channel shifts          channel\_shift\_range = 0.,          # set mode for filling points outside the input boundaries          fill\_mode ='nearest',          # value used for fill\_mode = "constant"          cval = 0.,          # randomly flip images          horizontal\_flip = True,          # randomly flip images          vertical\_flip = False,          # set rescaling factor (applied before any other transformation)          rescale = None,          # set function that will be applied on each input          preprocessing\_function = None,          # image data format, either "channels\_first" or "channels\_last"          data\_format = None,          # fraction of images reserved for validation (strictly between 0 and 1)          validation\_split = 0.0)        # Compute quantities required for featurewise normalization      # (std, mean, and principal components if ZCA whitening is applied).      datagen.fit(x\_train)        # Fit the model on the batches generated by datagen.flow().      model.fit\_generator(datagen.flow(x\_train, y\_train, batch\_size = batch\_size),                          validation\_data =(x\_test, y\_test),                          epochs = epochs, verbose = 1, workers = 4,                          callbacks = callbacks)    # Score trained model.  scores = model.evaluate(x\_test, y\_test, verbose = 1)  print('Test loss:', scores[0])  print('Test accuracy:', scores[1]) |

**Results & Conclusion:**   
On the ImageNet dataset,  the authors uses a 152-layers ResNet, which is 8 times more deep than VGG19 but still have less parameters. An ensemble of these ResNets generated an error of only 3.7% on ImageNet test set, the result which won ILSVRC 2015 competition. On COCO object detection dataset, it also generates a 28% relative improvement due to its very deep representation. 



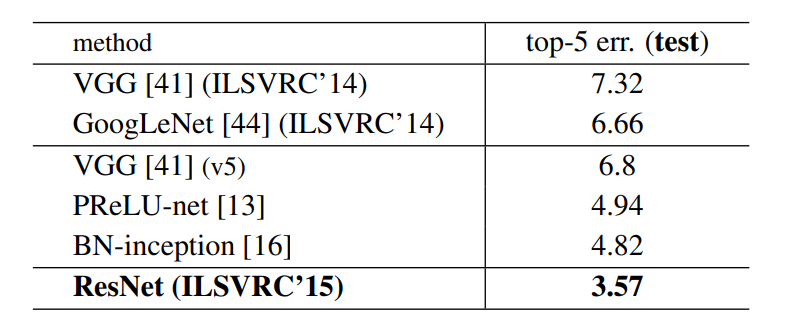
*Error-rate on ResNet Architecture*

* The result above shows that shortcut connections would be able to solve the problem caused by increasing the layers because as we increase layers from 18 to 34 the error rate on ImageNet Validation Set also decreases unlike the plain network.



*top-1 and top-5 Error rate on ImageNet Validation Set.*

* Below are the results on ImageNet Test Set. The *3.57%*top-5 error rate of ResNet was the lowest and thus ResNet architecture came first in ImageNet classification challenge in 2015.



1. What do Skip Connections entail?

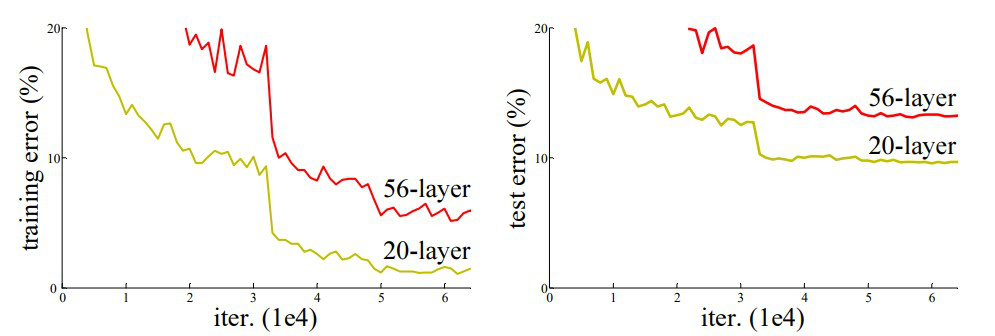
Skip connections are a technique that allows convolutional neural networks (CNNs) to bypass some layers and connect directly to deeper or shallower ones. They can improve the performance and efficiency of CNNs, but they also have some drawbacks and limitations. In this article, we will explore what skip connections are, how they work, and what are their benefits and drawbacks for CNNs

Skip connections are a type of shortcut that connects the output of one layer to the input of another layer that is not adjacent to it. For example, in a CNN with four layers, A, B, C, and D, a skip connection could connect layer A to layer C, or layer B to layer D, or both. Skip connections can be implemented in different ways, such as adding, concatenating, or multiplying the outputs of the skipped layers.

Skip connections work by allowing information and gradients to flow more easily through the network. Information is the input data and the features extracted by the layers, while gradients are the signals that adjust the weights of the layers during backpropagation. Skip connections can help to preserve information and gradients that might otherwise be lost or diluted by passing through multiple layers. They can also help to combine features from different levels of abstraction and resolution, which can enhance the representation power of the network.

1. What is the definition of a residual Block?

After the first CNN-based architecture (AlexNet) that win the ImageNet 2012 competition, Every subsequent winning architecture uses more layers in a deep neural network to reduce the error rate. This works for less number of layers, but when we increase the number of layers, there is a common problem in deep learning associated with that called the Vanishing/Exploding gradient. This causes the gradient to become 0 or too large. Thus when we increases number of layers, the training and test error rate also increases.

[](https://media.geeksforgeeks.org/wp-content/uploads/20200424200128/abc.jpg)

*Comparison of 20-layer vs 56-layer architecture*

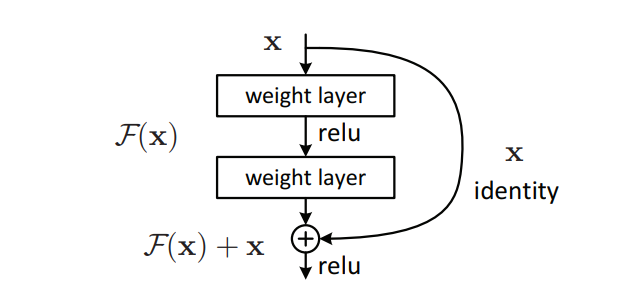
In the above plot, we can observe that a 56-layer CNN gives more error rate on both training and testing dataset than a 20-layer CNN architecture. After analyzing more on error rate the authors were able to reach conclusion that it is caused by vanishing/exploding gradient.

ResNet, which was proposed in 2015 by researchers at Microsoft Research introduced a new architecture called Residual Network.

**Residual Network:**In order to solve the problem of the vanishing/exploding gradient, this architecture introduced the concept called Residual Blocks. In this network, we use a technique called ***skip connections***. The skip connection connects activations of a  layer to further layers by skipping some layers in between. This forms a residual block. Resnets are made by stacking these residual blocks together.

The approach behind this network is instead of layers learning the underlying mapping, we allow the network to fit the residual mapping. So, instead of say H(x), initial mapping*,*let the network fit,

*F(x) := H(x) - x* which gives *H(x) := F(x) + x*.

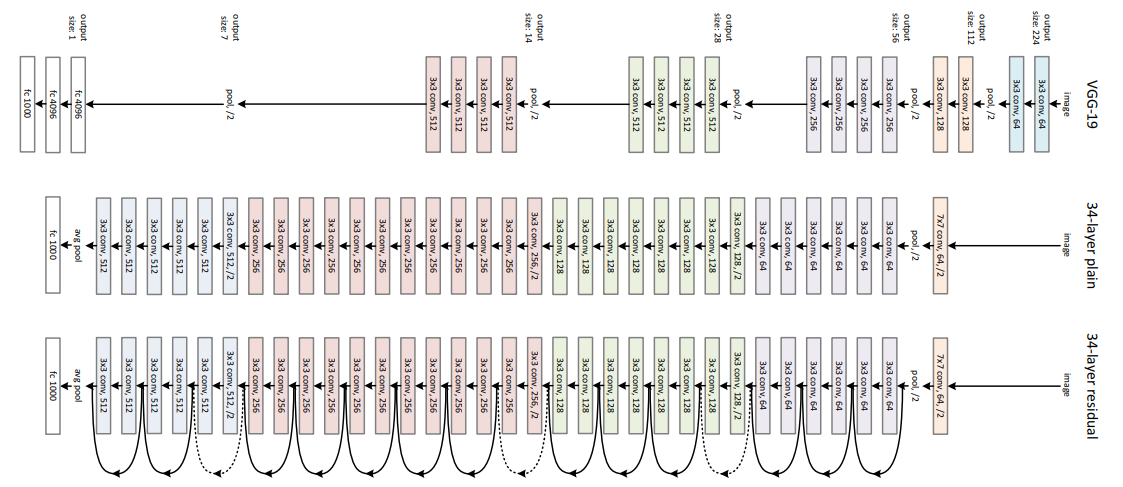
[](https://media.geeksforgeeks.org/wp-content/uploads/20200424011510/Residual-Block.PNG)

*Skip (Shortcut) connection*

The advantage of adding this type of skip connection is that if any layer hurt the performance of architecture then it will be skipped by regularization. So, this results in training a very deep neural network without the problems caused by vanishing/exploding gradient.  The authors of the paper experimented on 100-1000 layers of the CIFAR-10 dataset.

There is a similar approach called “highway networks”, these networks also use skip connection. Similar to LSTM these skip connections also use parametric gates. These gates determine how much information passes through the skip connection. This architecture however has not provided accuracy better than ResNet architecture.

**Network Architecture:** This network uses a 34-layer plain network architecture inspired by VGG-19 in which then the shortcut connection is added. These shortcut connections then convert the architecture into a residual network.  

[](https://media.geeksforgeeks.org/wp-content/uploads/20200424011138/ResNet.PNG)

*ResNet -34 architecture*

**Implementation:** Using the Tensorflow and Keras API, we can design ResNet architecture (including Residual Blocks) from scratch. Below is the implementation of different ResNet architecture. For this implementation, we use the CIFAR-10 dataset. This dataset contains 60, 000 32×32 color images in 10 different classes (airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks), etc. This dataset can be assessed from k*eras.datasets* API function.

**Step 1:** First, we import the keras module and its APIs. These APIs help in building the architecture of the ResNet model.

**Code:**Importing libraries

# Import Keras modules and its important APIs

import keras

from keras.layers import Dense, Conv2D, BatchNormalization, Activation

from keras.layers import AveragePooling2D, Input, Flatten

from keras.optimizers import Adam

from keras.callbacks import ModelCheckpoint, LearningRateScheduler

from keras.callbacks import ReduceLROnPlateau

from keras.preprocessing.image import ImageDataGenerator

from keras.regularizers import l2

from keras import backend as K

from keras.models import Model

from keras.datasets import cifar10

import numpy as np

import os

**Step 2:** Now, We set different hyper parameters that are required for ResNet architecture. We also did some preprocessing on our dataset to prepare it for training.

1. How can transfer learning help with problems?

We, humans, are very good at applying the transfer of knowledge between tasks. This means that whenever we encounter a new problem or a task. Similarly **Transfer learning is a smart method in machine learning where a model uses knowledge from one task to help with a different, but related, task**. Instead of learning from zero, the model uses what it already knows to solve new problems faster and better.

This is especially helpful when there isn’t much data available.**Transfer learning is making a big impact in areas like understanding language and recognizing images**. This article will explain what transfer learning is, its benefits, and how it’s changing various field.

**What is Transfer Learning?**

Transfer learning is a technique in [machine learning](https://www.geeksforgeeks.org/machine-learning/) where a model trained on one task is used as the starting point for a model on a second task. This can be useful when the second task is similar to the first task, or when there is limited data available for the second task. By using the learned features from the first task as a starting point, the model can learn more quickly and effectively on the second task. This can also help to prevent [overfitting](https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/" \t "_blank), as the model will have already learned general features that are likely to be useful in the second task.

**Why do we need Transfer Learning?**

Transfer learning is essential in machine learning for several reasons:

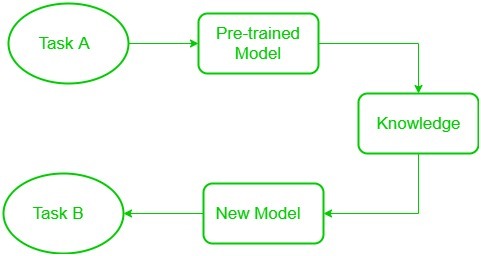
* **Limited Data**: In many real-world scenarios, obtaining a large amount of labeled data for training a model from scratch can be difficult and expensive. Transfer learning allows us to leverage pre-trained models and their knowledge, reducing the need for vast amounts of data.
* **Improved Performance**: By starting with a pre-trained model, which has already learned from a large dataset, we can achieve better performance on new tasks more quickly. This is especially useful in applications where accuracy and efficiency are crucial.
* **Time and Cost Efficiency**: Transfer learning saves time and resources because it speeds up the training process. Instead of training a new model from scratch, we can build on existing models and fine-tune them for specific tasks.
* **Adaptability**: Models trained on one task can be adapted to perform well on related tasks. This adaptability makes transfer learning suitable for a wide range of applications, from image recognition to natural language processing.

**How does Transfer Learning work?**

This is a general summary of how transfer learning works:

* **Pre-trained Model:** Start with a model that has previously been trained for a certain task using a large set of data. Frequently trained on extensive datasets, this model has identified general features and patterns relevant to numerous related jobs.
* **Base Model:** The model that has been pre-trained is known as the base model. It is made up of layers that have utilized the incoming data to learn hierarchical feature representations.
* **Transfer Layers:** In the pre-trained model, find a set of layers that capture generic information relevant to the new task as well as the previous one. Because they are prone to learning low-level information, these layers are frequently found near the top of the network.
* **Fine-tuning:** Using the dataset from the new challenge to retrain the chosen layers. We define this procedure as fine-tuning. The goal is to preserve the knowledge from the pre-training while enabling the model to modify its parameters to better suit the demands of the current assignment.

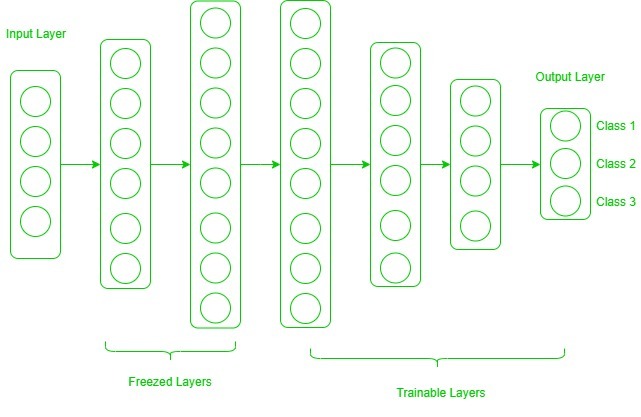
The Block diagram is shown below as follows:



*Transfer Learning*

**Low-level features** learned for task A should be beneficial for learning of model for task B.

**Freezed and Trainable Layers:**



In transfer learning, there are two main components: frozen layers and modifiable layers.

1. **Frozen Layers**: These are the layers of a pre-trained model that are kept unchanged during the fine-tuning process. Frozen layers retain the knowledge learned from the original task and are used to extract general features from the input data.
2. **Modifiable Layers**: These are the layers of the model that are adjusted or re-trained during fine-tuning. Modifiable layers learn task-specific features from the new dataset. By focusing on these layers, the model can adapt to the specific requirements of the new task.

Now, one may ask how to determine which layers we need to freeze, and which layers need to train. The answer is simple, the more you want to inherit features from a pre-trained model, the more you have to freeze layers.

Let’s consider all situations where the size and dataset of the target task vary from the base network.

* **The target dataset is small and similar to the base network dataset:** Since the target dataset is small, that means we can fine-tune the pre-trained network with the target dataset. But this may lead to a problem of overfitting. Also, there may be some changes in the number of classes in the target task. So, in such a case we remove the fully connected layers from the end, maybe one or two, and add a new fully connected layer satisfying the number of new classes. Now, we freeze the rest of the model and only train newly added layers.
* **The target dataset is large and similar to the base training dataset:** In such cases when the dataset is large, and it can hold a pre-trained model there will be no chance of overfitting. Here, also the last full-connected layer is removed, and a new fully-connected layer is added with the proper number of classes. Now, the entire model is trained on a new dataset. This makes sure to tune the model on a new large dataset keeping the model architecture the same.
* **The target dataset is small and different from the base network dataset:** Since the target dataset is different, using high-level features of the pre-trained model will not be useful. In such a case, remove most of the layers from the end in a pre-trained model, and add new layers a satisfying number of classes in a new dataset. This way we can use low-level features from the pre-trained model and train the rest of the layers to fit a new dataset. Sometimes, it is beneficial to train the entire network after adding a new layer at the end.
* **The target dataset is large and different from the base network dataset:**Since the target network is large and different, the best way is to remove the last layers from the pre-trained network and add layers with a satisfying number of classes, then train the entire network without freezing any layer.

Transfer learning is a very effective and fast way, to begin with, a problem. It gives the direction to move, and most of the time best results are also obtained by transfer learning.

1. What is transfer learning, and how does it work?

We, humans, are very good at applying the transfer of knowledge between tasks. This means that whenever we encounter a new problem or a task. Similarly **Transfer learning is a smart method in machine learning where a model uses knowledge from one task to help with a different, but related, task**. Instead of learning from zero, the model uses what it already knows to solve new problems faster and better.

This is especially helpful when there isn’t much data available.**Transfer learning is making a big impact in areas like understanding language and recognizing images**. This article will explain what transfer learning is, its benefits, and how it’s changing various field.

**What is Transfer Learning?**

Transfer learning is a technique in [machine learning](https://www.geeksforgeeks.org/machine-learning/) where a model trained on one task is used as the starting point for a model on a second task. This can be useful when the second task is similar to the first task, or when there is limited data available for the second task. By using the learned features from the first task as a starting point, the model can learn more quickly and effectively on the second task. This can also help to prevent [overfitting](https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/" \t "_blank), as the model will have already learned general features that are likely to be useful in the second task.

**Why do we need Transfer Learning?**

Transfer learning is essential in machine learning for several reasons:

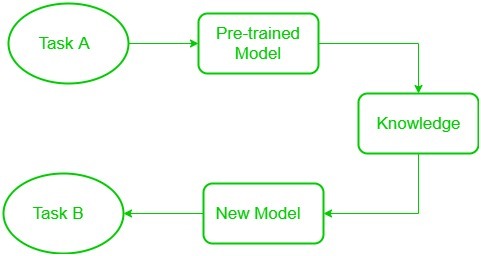
* **Limited Data**: In many real-world scenarios, obtaining a large amount of labeled data for training a model from scratch can be difficult and expensive. Transfer learning allows us to leverage pre-trained models and their knowledge, reducing the need for vast amounts of data.
* **Improved Performance**: By starting with a pre-trained model, which has already learned from a large dataset, we can achieve better performance on new tasks more quickly. This is especially useful in applications where accuracy and efficiency are crucial.
* **Time and Cost Efficiency**: Transfer learning saves time and resources because it speeds up the training process. Instead of training a new model from scratch, we can build on existing models and fine-tune them for specific tasks.
* **Adaptability**: Models trained on one task can be adapted to perform well on related tasks. This adaptability makes transfer learning suitable for a wide range of applications, from image recognition to natural language processing.

**How does Transfer Learning work?**

This is a general summary of how transfer learning works:

* **Pre-trained Model:** Start with a model that has previously been trained for a certain task using a large set of data. Frequently trained on extensive datasets, this model has identified general features and patterns relevant to numerous related jobs.
* **Base Model:** The model that has been pre-trained is known as the base model. It is made up of layers that have utilized the incoming data to learn hierarchical feature representations.
* **Transfer Layers:** In the pre-trained model, find a set of layers that capture generic information relevant to the new task as well as the previous one. Because they are prone to learning low-level information, these layers are frequently found near the top of the network.
* **Fine-tuning:** Using the dataset from the new challenge to retrain the chosen layers. We define this procedure as fine-tuning. The goal is to preserve the knowledge from the pre-training while enabling the model to modify its parameters to better suit the demands of the current assignment.

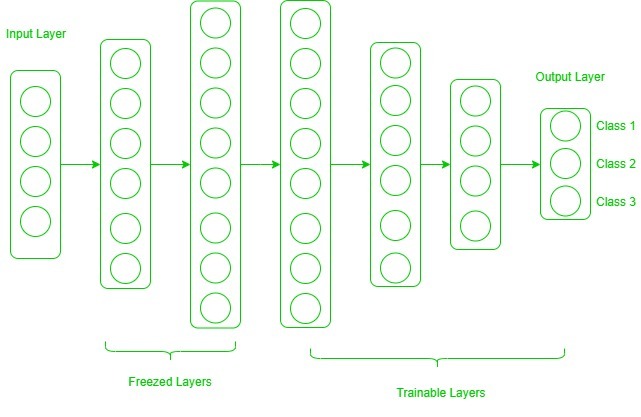
The Block diagram is shown below as follows:



*Transfer Learning*

**Low-level features** learned for task A should be beneficial for learning of model for task B.

**Freezed and Trainable Layers:**



In transfer learning, there are two main components: frozen layers and modifiable layers.

1. **Frozen Layers**: These are the layers of a pre-trained model that are kept unchanged during the fine-tuning process. Frozen layers retain the knowledge learned from the original task and are used to extract general features from the input data.
2. **Modifiable Layers**: These are the layers of the model that are adjusted or re-trained during fine-tuning. Modifiable layers learn task-specific features from the new dataset. By focusing on these layers, the model can adapt to the specific requirements of the new task.

Now, one may ask how to determine which layers we need to freeze, and which layers need to train. The answer is simple, the more you want to inherit features from a pre-trained model, the more you have to freeze layers.

Let’s consider all situations where the size and dataset of the target task vary from the base network.

* **The target dataset is small and similar to the base network dataset:** Since the target dataset is small, that means we can fine-tune the pre-trained network with the target dataset. But this may lead to a problem of overfitting. Also, there may be some changes in the number of classes in the target task. So, in such a case we remove the fully connected layers from the end, maybe one or two, and add a new fully connected layer satisfying the number of new classes. Now, we freeze the rest of the model and only train newly added layers.
* **The target dataset is large and similar to the base training dataset:** In such cases when the dataset is large, and it can hold a pre-trained model there will be no chance of overfitting. Here, also the last full-connected layer is removed, and a new fully-connected layer is added with the proper number of classes. Now, the entire model is trained on a new dataset. This makes sure to tune the model on a new large dataset keeping the model architecture the same.
* **The target dataset is small and different from the base network dataset:** Since the target dataset is different, using high-level features of the pre-trained model will not be useful. In such a case, remove most of the layers from the end in a pre-trained model, and add new layers a satisfying number of classes in a new dataset. This way we can use low-level features from the pre-trained model and train the rest of the layers to fit a new dataset. Sometimes, it is beneficial to train the entire network after adding a new layer at the end.
* **The target dataset is large and different from the base network dataset:**Since the target network is large and different, the best way is to remove the last layers from the pre-trained network and add layers with a satisfying number of classes, then train the entire network without freezing any layer.

Transfer learning is a very effective and fast way, to begin with, a problem. It gives the direction to move, and most of the time best results are also obtained by transfer learning.

* 1. HOW DO NEURAL NETWORKS LEARN FEATURES?

## What is a neural network?

A neural network is a method in artificial intelligence that teaches computers to process data in a way that is inspired by the human brain. It is a type of machine learning process, called deep learning, that uses interconnected nodes or neurons in a layered structure that resembles the human brain. It creates an adaptive system that computers use to learn from their mistakes and improve continuously. Thus, artificial neural networks attempt to solve complicated problems, like summarizing documents or recognizing faces, with greater accuracy.

## Why are neural networks important?

Neural networks can help computers make intelligent decisions with limited human assistance. This is because they can learn and model the relationships between input and output data that are nonlinear and complex. For instance, they can do the following tasks.

### Make generalizations and inferences

Neural networks can comprehend unstructured data and make general observations without explicit training. For instance, they can recognize that two different input sentences have a similar meaning:

* Can you tell me how to make the payment?
* How do I transfer money?

A neural network would know that both sentences mean the same thing. Or it would be able to broadly recognize that Baxter Road is a place, but Baxter Smith is a person’s name.

## What are neural networks used for?

Neural networks have several use cases across many industries, such as the following:

* Medical diagnosis by medical image classification
* Targeted marketing by social network filtering and behavioral data analysis
* Financial predictions by processing historical data of financial instruments
* Electrical load and energy demand forecasting
* Process and quality control
* Chemical compound identification

We give four of the important applications of neural networks below.

### Computer vision

Computer vision is the ability of computers to extract information and insights from images and videos. With neural networks, computers can distinguish and recognize images similar to humans. Computer vision has several applications, such as the following:

* Visual recognition in self-driving cars so they can recognize road signs and other road users
* Content moderation to automatically remove unsafe or inappropriate content from image and video archives
* Facial recognition to identify faces and recognize attributes like open eyes, glasses, and facial hair
* Image labeling to identify brand logos, clothing, safety gear, and other image details

### Speech recognition

Neural networks can analyze human speech despite varying speech patterns, pitch, tone, language, and accent. Virtual assistants like Amazon Alexa and automatic transcription software use speech recognition to do tasks like these:

* Assist call center agents and automatically classify calls
* Convert clinical conversations into documentation in real time
* Accurately subtitle videos and meeting recordings for wider content reach

### Natural language processing

Natural language processing (NLP) is the ability to process natural, human-created text. Neural networks help computers gather insights and meaning from text data and documents. NLP has several use cases, including in these functions:

* Automated virtual agents and chatbots
* Automatic organization and classification of written data
* Business intelligence analysis of long-form documents like emails and forms
* Indexing of key phrases that indicate sentiment, like positive and negative comments on social media
* Document summarization and article generation for a given topic

### Recommendation engines

Neural networks can track user activity to develop personalized recommendations. They can also analyze all user behavior and discover new products or services that interest a specific user. For example, Curalate, a Philadelphia-based startup, helps brands convert social media posts into sales. Brands use Curalate’s intelligent product tagging (IPT) service to automate the collection and curation of user-generated social content. IPT uses neural networks to automatically find and recommend products relevant to the user’s social media activity. Consumers don't have to hunt through online catalogs to find a specific product from a social media image. Instead, they can use Curalate’s auto product tagging to purchase the product with ease.

## How do neural networks work?

The human brain is the inspiration behind neural network architecture. Human brain cells, called neurons, form a complex, highly interconnected network and send electrical signals to each other to help humans process information. Similarly, an artificial neural network is made of artificial neurons that work together to solve a problem. Artificial neurons are software modules, called nodes, and artificial neural networks are software programs or algorithms that, at their core, use computing systems to solve mathematical calculations.

### Simple neural network architecture

A basic neural network has interconnected artificial neurons in three layers:

#### Input Layer

Information from the outside world enters the artificial neural network from the input layer. Input nodes process the data, analyze or categorize it, and pass it on to the next layer.

#### Hidden Layer

Hidden layers take their input from the input layer or other hidden layers. Artificial neural networks can have a large number of hidden layers. Each hidden layer analyzes the output from the previous layer, processes it further, and passes it on to the next layer.

#### Output Layer

The output layer gives the final result of all the data processing by the artificial neural network. It can have single or multiple nodes. For instance, if we have a binary (yes/no) classification problem, the output layer will have one output node, which will give the result as 1 or 0. However, if we have a multi-class classification problem, the output layer might consist of more than one output node.

### Deep neural network architecture

Deep neural networks, or deep learning networks, have several hidden layers with millions of artificial neurons linked together. A number, called weight, represents the connections between one node and another. The weight is a positive number if one node excites another, or negative if one node suppresses the other. Nodes with higher weight values have more influence on the other nodes.  
Theoretically, deep neural networks can map any input type to any output type. However, they also need much more training as compared to other machine learning methods. They need millions of examples of training data rather than perhaps the hundreds or thousands that a simpler network might need.

## What are the types of neural networks?

Artificial neural networks can be categorized by how the data flows from the input node to the output node. Below are some examples:

### Feedforward neural networks

Feedforward neural networks process data in one direction, from the input node to the output node. Every node in one layer is connected to every node in the next layer. A feedforward network uses a feedback process to improve predictions over time.

### Backpropagation algorithm

Artificial neural networks learn continuously by using corrective feedback loops to improve their predictive analytics. In simple terms, you can think of the data flowing from the input node to the output node through many different paths in the neural network. Only one path is the correct one that maps the input node to the correct output node. To find this path, the neural network uses a feedback loop, which works as follows:

1. Each node makes a guess about the next node in the path.
2. It checks if the guess was correct. Nodes assign higher weight values to paths that lead to more correct guesses and lower weight values to node paths that lead to incorrect guesses.
3. For the next data point, the nodes make a new prediction using the higher weight paths and then repeat Step 1.

### Convolutional neural networks

The hidden layers in convolutional neural networks perform specific mathematical functions, like summarizing or filtering, called convolutions. They are very useful for image classification because they can extract relevant features from images that are useful for image recognition and classification. The new form is easier to process without losing features that are critical for making a good prediction. Each hidden layer extracts and processes different image features, like edges, color, and depth.

## How to train neural networks?

Neural network training is the process of teaching a neural network to perform a task. Neural networks learn by initially processing several large sets of labeled or unlabeled data. By using these examples, they can then process unknown inputs more accurately.

### Supervised learning

In supervised learning, data scientists give artificial neural networks labeled datasets that provide the right answer in advance. For example, a deep learning network training in facial recognition initially processes hundreds of thousands of images of human faces, with various terms related to ethnic origin, country, or emotion describing each image.

The neural network slowly builds knowledge from these datasets, which provide the right answer in advance. After the network has been trained, it starts making guesses about the ethnic origin or emotion of a new image of a human face that it has never processed before.

## What is deep learning in the context of neural networks?

Artificial intelligence is the field of computer science that researches methods of giving machines the ability to perform tasks that require human intelligence. Machine learning is an artificial intelligence technique that gives computers access to very large datasets and teaches them to learn from this data. Machine learning software finds patterns in existing data and applies those patterns to new data to make intelligent decisions. Deep learning is a subset of machine learning that uses deep learning networks to process data.

### Machine learning vs. deep learning

Traditional machine learning methods require human input for the machine learning software to work sufficiently well. A data scientist manually determines the set of relevant features that the software must analyze. This limits the software’s ability, which makes it tedious to create and manage.

On the other hand, in deep learning, the data scientist gives only raw data to the software. The deep learning network derives the features by itself and learns more independently. It can analyze unstructured datasets like text documents, identify which data attributes to prioritize, and solve more complex problems.

For example, if you were training a machine learning software to identify an image of a pet correctly, you would need to take these steps:

* Find and label thousands of pet images, like cats, dogs, horses, hamsters, parrots, and so on, manually.
* Tell the machine learning software what features to look for so it can identify the image using elimination. For instance, it might count the number of legs, then check for eye shape, ear shape, tail, fur, and so on.
* Manually assess and change the labeled datasets to improve the software’s accuracy. For example, if your training set has too many pictures of black cats, the software will correctly identify a black cat but not a white one.
* In deep learning, however, the neural networks would process all the images and automatically determine that they need to analyze the number of legs and the face shape first, then look at the tails last to correctly identify the animal in the image.
  1. WHY IS FINE-TUNING BETTER THAN START-UP TRAINING?

### Training

At the onset of training (or, in this context, pre-training), the model has not yet “learned” anything. Training begins with a random initialization of model parameters—the varying weights and biases applied to the mathematical operations occurring at each node in the [neural network](https://www.ibm.com/topics/neural-networks).

Training occurs iteratively in two phases: in a forward pass, the model makes predictions for a batch of sample inputs from the training dataset, and a loss function measures the difference (or loss) between the model’s predictions for each input and the “correct” answers (or ground truth); during backpropagation, an optimization algorithm—typically [gradient descent](https://www.ibm.com/topics/gradient-descent)—is used to adjust model weights across the network to reduce loss. These adjustments to model weights are how the model “learns.” The process is repeated across multiple training epochs until the model is deemed to be sufficiently trained.

Conventional [supervised learning](https://www.ibm.com/topics/supervised-learning), which is typically used to pre-train models for computer vision tasks like image classification, object detection or [image segmentation](https://www.ibm.com/topics/image-segmentation), uses labeled data: labels (or annotations) provide both the range of possible answers and the ground truth output for each sample.

LLMs are typically pre-trained through [self-supervised learning (SSL)](https://www.ibm.com/topics/self-supervised-learning), in which models learn through pretext tasks that are designed to derive ground truth from the inherent structure of unlabeled data. These pretext tasks impart knowledge useful for downstream tasks. They typically take one of two approaches:

* Self-prediction: masking some part of the original input and tasking the model with reconstructing it. This is the dominant mode of training for LLMs.
* Contrastive learning: training models to learn similar embeddings for related inputs and different embeddings for unrelated inputs. This is used prominently in computer vision models designed for [few-shot](https://www.ibm.com/topics/few-shot-learning) or [zero-shot learning](https://www.ibm.com/topics/zero-shot-learning), like Contrasting Language-Image Pretraining (CLIP).

SSL thus allows for the use of massively large datasets in training without the burden of having to annotate millions or billions of data points. This saves a tremendous amount of labor, but nevertheless requires huge computational resources.

### Fine-tuning

Conversely, fine-tuning entails techniques to further train a model whose weights have already been updated through prior training. Using the base model’s previous knowledge as a starting point, fine-tuning tailors the model by training it on a smaller, task-specific dataset.

While that task-specific dataset could theoretically have been used for the initial training, training a large model from scratch on a small dataset risks [overfitting](https://www.ibm.com/topics/overfitting): the model might learn to perform well on the training examples, but generalize poorly to new data. This would render the model ill-suited to its given task and defeat the purpose of model training.

Fine-tuning thus provides the best of both worlds: leveraging the broad knowledge and stability gained from pre-training on a massive set of data and honing the model’s understanding of more detailed, specific concepts. Given the increasing prowess of open source foundation models, the benefits can often be enjoyed without any of the financial, computational or logistical burden of pre-training.