



RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Machine Learning**

Subject Code: **CS-601**

Semester: **6th**



LIKE & FOLLOW US ON FACEBOOK

facebook.com/rgpvnotes.in

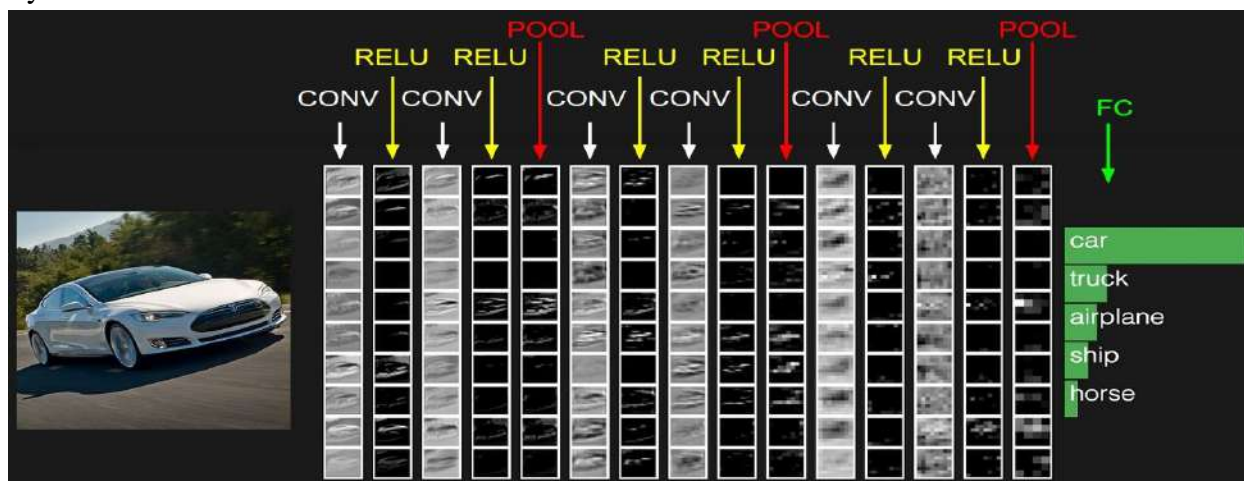
Unit –III

Convolutional neural network, flattening, subsampling, padding, stride, convolution layer, pooling layer, loss layer, dense layer 1x1 convolution, inception network, input channels, transfer learning, one shot learning, dimension reductions, implementation of CNN like tensor flow, keras etc

1. Convolutional Neural Networks (CNNs / ConvNets)

Convolutional Neural Networks are very similar to ordinary Neural Networks from the previous chapter: they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer and all the tips/tricks we developed for learning regular Neural Networks still apply.

So what changes? ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network. As we described above, a simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We use three main types of layers to build ConvNet architectures: **Convolutional Layer**, **Pooling Layer**, and **Fully-Connected Layer** (exactly as seen in regular Neural Networks). We will stack these layers to form a full ConvNet



Architecture.

Example Architecture: Overview. We will go into more details below, but a simple ConvNet for CIFAR-10 classification could have the architecture [INPUT - CONV - RELU - POOL - FC]. In more detail:

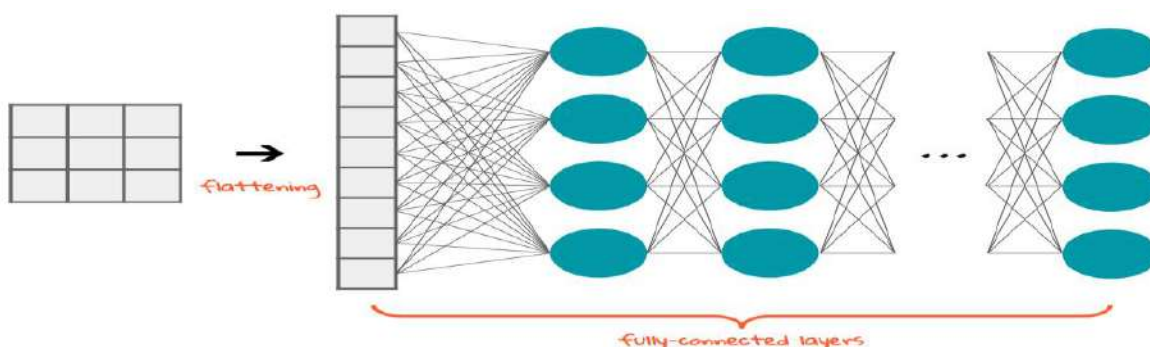
- INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.

- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as $[32 \times 32 \times 12]$ if we decided to use 12 filters.
- RELU layer will apply an element wise activation function, such as the $\max(0, x)$ thresholding at zero. This leaves the size of the volume unchanged ($[32 \times 32 \times 12]$).
- POOL layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as $[16 \times 16 \times 12]$.
- FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size $[1 \times 1 \times 10]$, where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

In this way, ConvNets transform the original image layer by layer from the original pixel values to the final class scores. Note that some layers contain parameters and other don't. In particular, the CONV/FC layers perform transformations that are a function of not only the activations in the input volume, but also of the parameters (the weights and biases of the neurons). On the other hand, the RELU/POOL layers will implement a fixed function. The parameters in the CONV/FC layers will be trained with gradient descent so that the class scores that the ConvNet computes are consistent with the labels in the training set for each image.

2. Flattening

Flattening is converting the data into a 1-dimensional array for inputting it to the next layer. We flatten the output of the convolutional layers to create a single long feature vector. And it is connected to the final classification model, which is called a **fully-connected** layer. In other words, we put all the pixel data in one line and make connections with the final layer. And once again

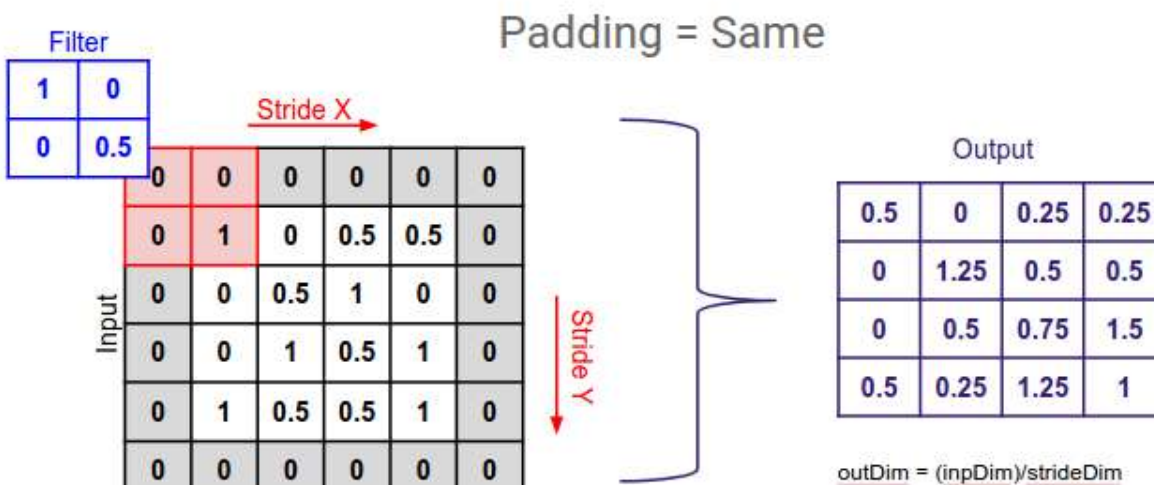


3. Subsampling (Supersample)

- Data Set is the entire collection of data to be analyzed. For inferential purposes, this may be treated as having been sampled from a population. All of the data set items will be classified by the process.
- Supersample is a subset of the data set chosen by simple random sampling. In our examples, it is the entire data set, but for larger data sets it will be considerably smaller. All computations prior to the final classification are performed on the supersample. For problems in moderate dimension (up to 50), the supersample will never need to be larger than 100,000–1,000,000 points, since the estimation error in a sample of this size is already too small to matter.
- Sample is one of several ($R_s R_s$) of size $N_s N_s$ chosen by simple random sampling from the supersample. All intensive search operations are conducted in the sample so that the supersample is only used for one iteration from the best solution found in the sample. The sample size $N_s N_s$ should be chosen to be large enough to reflect the essential structure of the data, while being small enough to keep the computations feasible.
- Subsample is one of several ($R_r R_r$) of size $N_r N_r$ chosen by simple random sampling from the sample that is used to begin iterations on the sample. This number should be very small because great diversity in starting points generates diversity in solutions, and increases the chance of finding the best local maximum of the likelihood.

4. What is Padding in Machine Learning?

Padding is a term relevant to convolutional neural networks as it refers to the amount of pixels added to an image when it is being processed by the kernel of a CNN. For example, if the padding in a CNN is set to zero, then every pixel value that is added will be of value zero. If, however, the zero padding is set to one, there will be a one pixel border added to the image with a pixel value of zero.



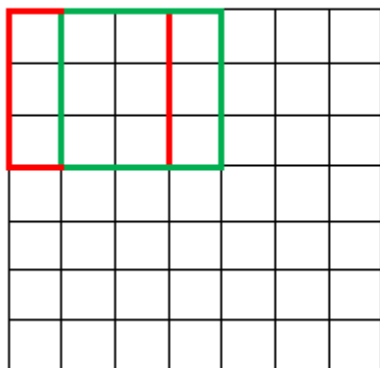
How does Padding work?

Padding works by extending the area of which a convolutional neural network processes an image. The kernel is the neural networks filter which moves across the image, scanning each pixel and converting the data into a smaller, or sometimes larger, format. In order to assist the kernel with processing the image, padding is added to the frame of the image to allow for more space for the kernel to cover the image. Adding padding to an image processed by a CNN allows for more accurate analysis of images.

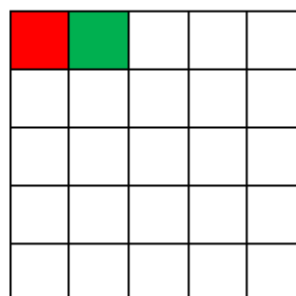
5. What is Stride (Machine Learning)?

Stride is a component of convolutional neural networks, or neural networks tuned for the compression of images and video data. Stride is a parameter of the neural network's filter that modifies the amount of movement over the image or video. For example, if a neural network's stride is set to 1, the filter will move one pixel, or unit, at a time. The size of the filter affects the encoded output volume, so stride is often set to a whole integer, rather than a fraction or decimal.

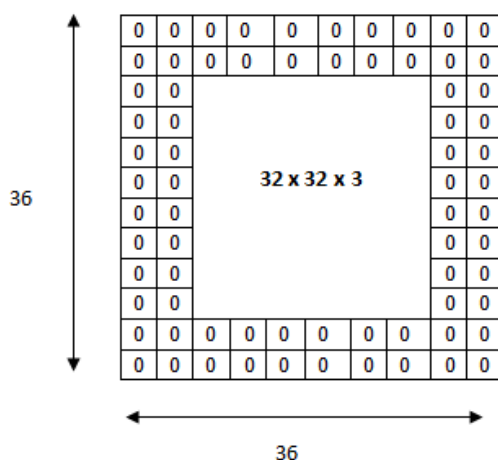
7 x 7 Input Volume



5 x 5 Output Volume



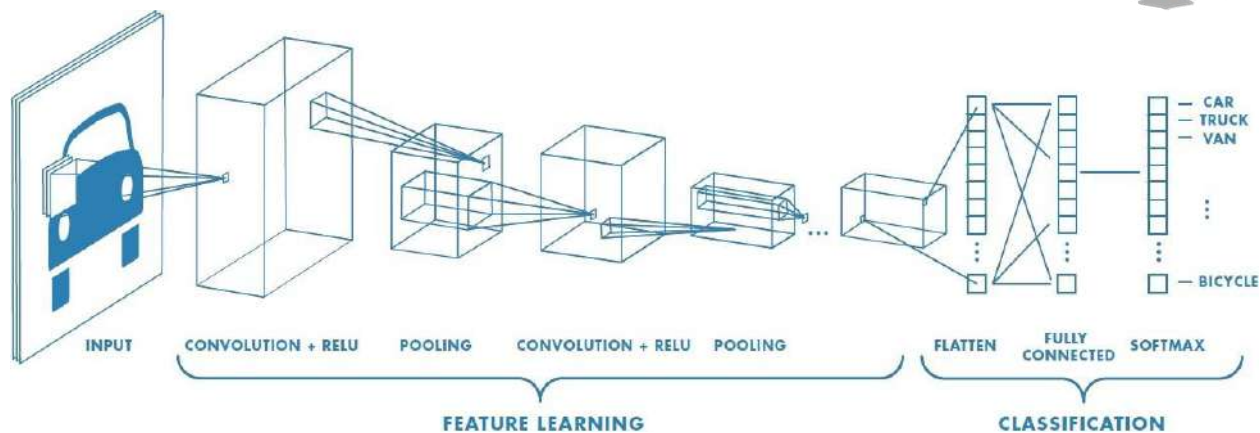
Imagine a convolutional neural network is taking an image and analyzing the content. If the filter size is 3x3 pixels, the contained nine pixels will be converted down to 1 pixel in the output layer. Naturally, as the stride, or movement, is increased, the resulting output will be smaller. Stride is a parameter that works in conjunction with [padding](#), the feature that adds blank, or empty pixels to the frame of the image to allow for a minimized reduction of size in the output layer. Roughly, it is a way of increasing the size of an image, to counteract the fact that stride reduces the size. Padding and stride are the foundational parameters of any convolutional neural network.



The input volume is 32 x 32 x 3. If we imagine two borders of zeros around the volume, this gives us a 36 x 36 x 3 volume. Then, when we apply our conv layer with our three 5 x 5 x 3 filters and a stride of 1, then we will also get a 32 x 32 x 3 output volume.

In neural networks, Convolutional neural network (ConvNets or CNNs) is one of the main categories to do images recognition, images classifications. Objects detections, recognition faces etc., are some of the areas where CNNs are widely used. CNN image classifications takes an input image, process it and classify it under certain categories (Eg., Dog, Cat, Tiger, Lion). Computers sees an input image as array of pixels and it depends on the image resolution. Based on the image resolution, it will see $h \times w \times d$ (h = Height, w = Width, d = Dimension). Eg., An image of 6 x 6 x 3 array of matrix of RGB (3 refers to RGB values) and an image of 4 x 4 x 1 array of matrix of grayscale image.

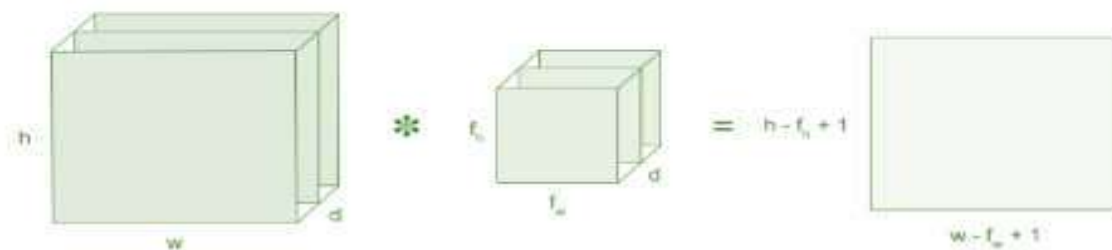
Technically, deep learning CNN models to train and test, each input image will pass it through a series of convolution layers with filters (Kernels), Pooling, fully connected layers (FC) and apply Softmax function to classify an object with probabilistic values between 0 and 1. The below figure is a complete flow of CNN to process an input image and classifies the objects based on values.



6. Convolution Layer

Convolution is the first layer to extract features from an input image. Convolution preserves the relationship between pixels by learning image features using small squares of input data. It is a mathematical operation that takes two inputs such as image matrix and a filter or kernel.

- An image matrix (volume) of dimension **$(h \times w \times d)$**
- A filter **$(f_h \times f_w \times d)$**
- Outputs a volume dimension **$(h - f_h + 1) \times (w - f_w + 1) \times 1$**



Consider a 5 x 5 whose image pixel values are 0, 1 and filter matrix 3 x 3 as shown in below

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

5 x 5 – Image Matrix

*

1	0	1
0	1	0
1	0	1

3 x 3 – Filter Matrix

Then the convolution of 5 x 5 image matrix multiplies with 3 x 3 filter matrix which is called “**Feature Map**” as output shown in below








1 _{x=1}	1 _{x=0}	1 _{x=1}	0	0
0 _{x=0}	1 _{x=1}	1 _{x=0}	1	0
0 _{x=1}	0 _{x=0}	1 _{x=1}	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved
Feature

Convolution of an image with different filters can perform operations such as edge detection, blur and sharpen by applying filters. The below example shows various convolution image after applying different types of filters (Kernels).

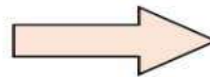
Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Strides

Stride is the number of pixels shifts over the input matrix. When the stride is 1 then we move the filters to 1 pixel at a time. When the stride is 2 then we move the filters to 2 pixels at a time and so on. The below figure shows convolution would work with a stride of 2.

1	2	3	4	5	6	7
11	12	13	14	15	16	17
21	22	23	24	25	26	27
31	32	33	34	35	36	37
41	42	43	44	45	46	47
51	52	53	54	55	56	57
61	62	63	64	65	66	67
71	72	73	74	75	76	77

Convolve with 3x3
filters filled with ones



108	126	
288	306	

Padding

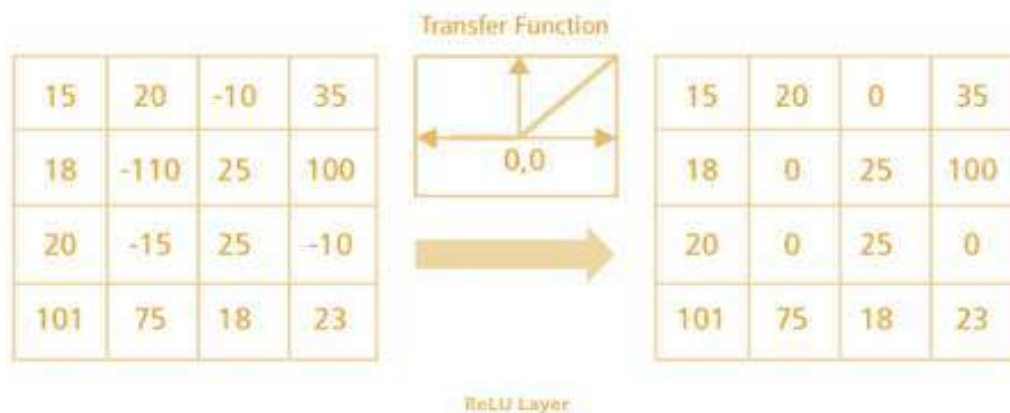
Sometimes filter does not fit perfectly fit the input image. We have two options:

- Pad the picture with zeros (zero-padding) so that it fits
- Drop the part of the image where the filter did not fit. This is called valid padding which keeps only valid part of the image.

Non Linearity (ReLU)

ReLU stands for Rectified Linear Unit for a non-linear operation. The output is $f(x) = \max(0, x)$.

Why ReLU is important : ReLU's purpose is to introduce non-linearity in our ConvNet. Since, the real world data would want our ConvNet to learn would be non-negative linear values.



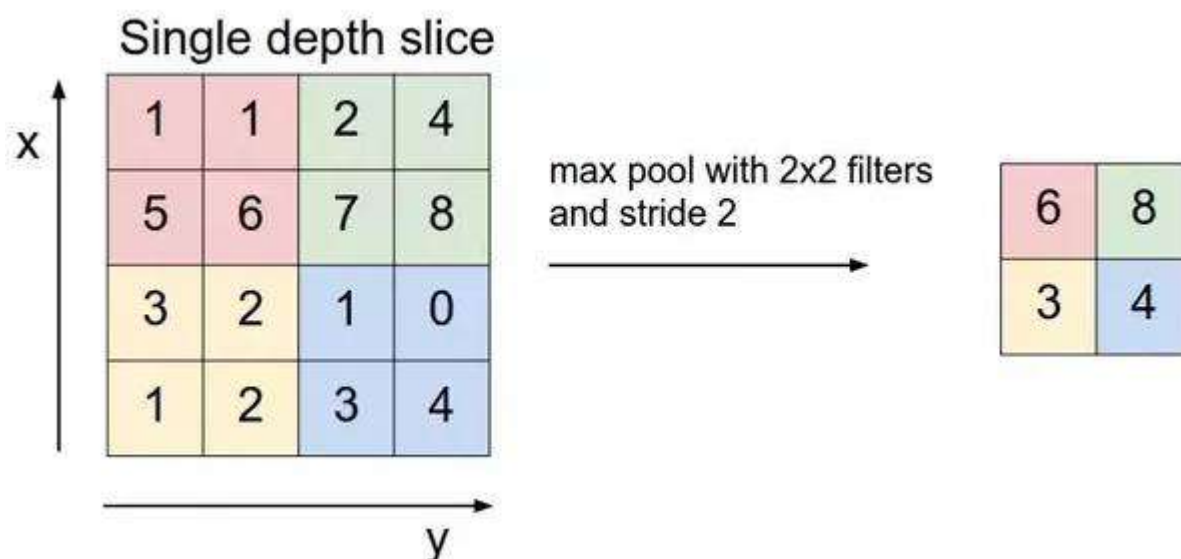
There are other non linear functions such as tanh or sigmoid that can also be used instead of ReLU. Most of the data scientists use ReLU since performance wise ReLU is better than the other two.

7. Pooling Layer

Pooling layers section would reduce the number of parameters when the images are too large. Spatial pooling also called subsampling or downsampling which reduces the dimensionality of each map but retains important information. Spatial pooling can be of different types:

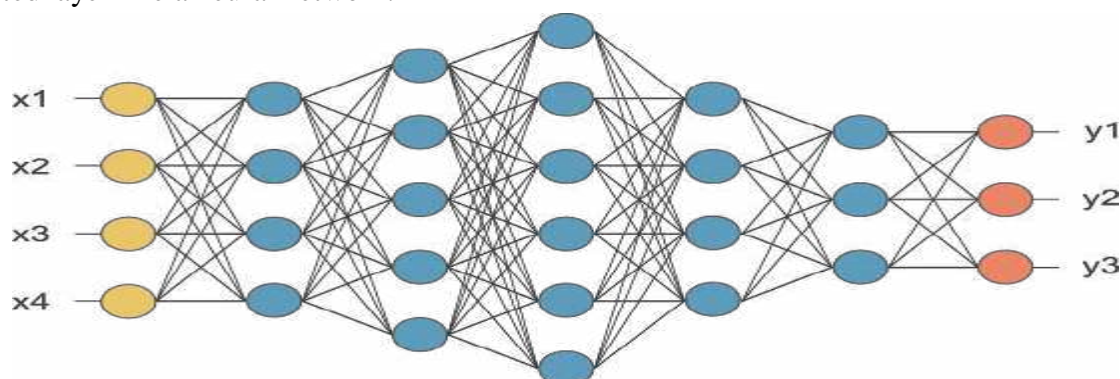
- Max Pooling
- Average Pooling
- Sum Pooling

Max pooling takes the largest element from the rectified feature map. Taking the largest element could also take the average pooling. Sum of all elements in the feature map call as sum pooling.

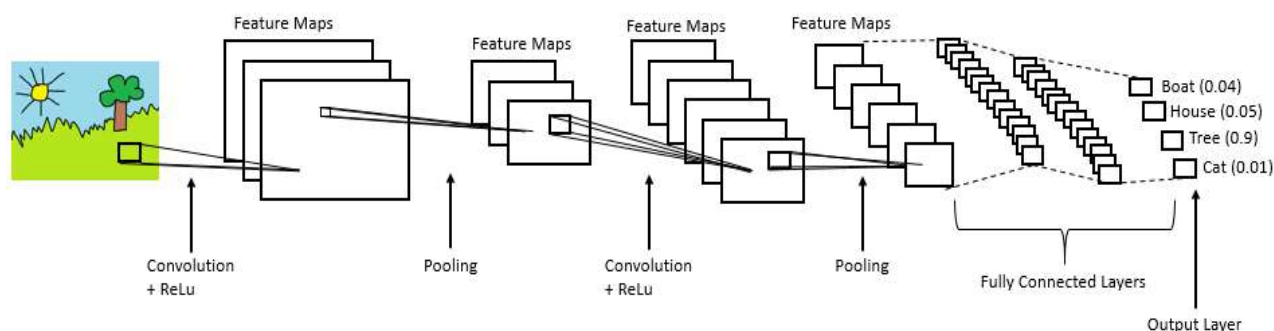


Fully Connected Layer

The layer we call as FC layer, we flattened our matrix into vector and feed it into a fully connected layer like a neural network.



In the above diagram, the feature map matrix will be converted as vector (x_1, x_2, x_3, \dots). With the fully connected layers, we combined these features together to create a model. Finally, we have an activation function such as softmax or sigmoid to classify the outputs as cat, dog, car, truck etc.,



Summary

- Provide input image into convolution layer
- Choose parameters, apply filters with strides, padding if requires. Perform convolution on the image and apply ReLU activation to the matrix.
- Perform pooling to reduce dimensionality size
- Add as many convolutional layers until satisfied
- Flatten the output and feed into a fully connected layer (FC Layer)
- Output the class using an activation function (Logistic Regression with cost functions) and classifies images.

In the next post, I would like to talk about some popular CNN architectures such as AlexNet, VGGNet, GoogLeNet, and ResNet.

8. **loss layer** : (Missing)

9. 1x1 convolutions

Convolutions layers are lighter than fully connected ones. But they still connect every input channels with every output channels for every position in the kernel windows. This is what

gives the $c_{in} * c_{out}$ multiplicative factor in the number of weights.

ILSVRC's Convnets use a lot of channels. 512 channels are used in VGG16's convolutional layers for example.

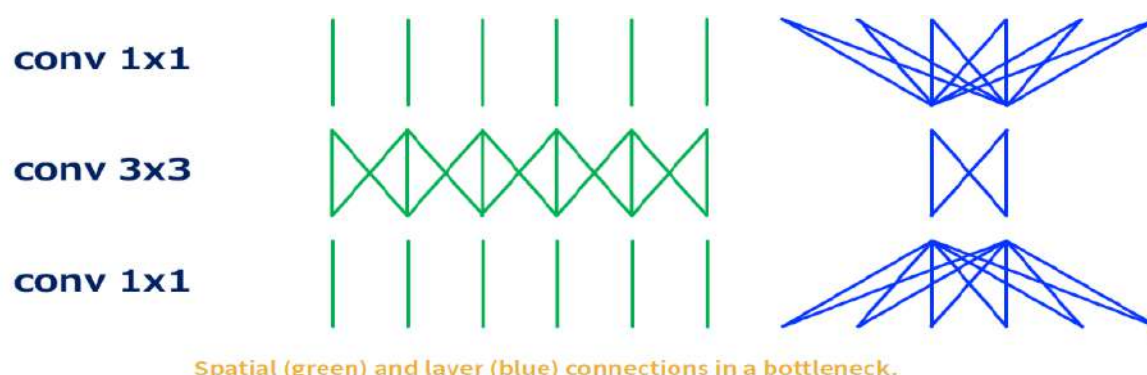
1x1 convolution is a solution to compensate for this.

It obviously doesn't bring anything at the spatial level: the kernel acts on one pixel at a time. But it acts as a fully connected layer pixel-wise.

We would usually have a 3x3 kernel size with 256 input and output channels. Instead of this, we first do a 1x1 convolutional layer bringing the number of channels down to something like 32. Then we perform the convolution with a 3x3 kernel size. We finally make another 1x1 convolutional layer to have 256 channels again.

The first solution needs $3^2 * 256^2 = 65,536$ weights. The second one needs $1^2 * 256 * 32 + 3^2 * 32^2 + 1^2 * 32 * 256 = 25,600$ weights. The convolution kernel is more than 2 times lighter.

A 1x1 convolution kernel acts as an embedding solution. It reduces the size of the input vector, the number of channels. It makes it more meaningful. The 1x1 convolutional layer is also called a Pointwise Convolution.

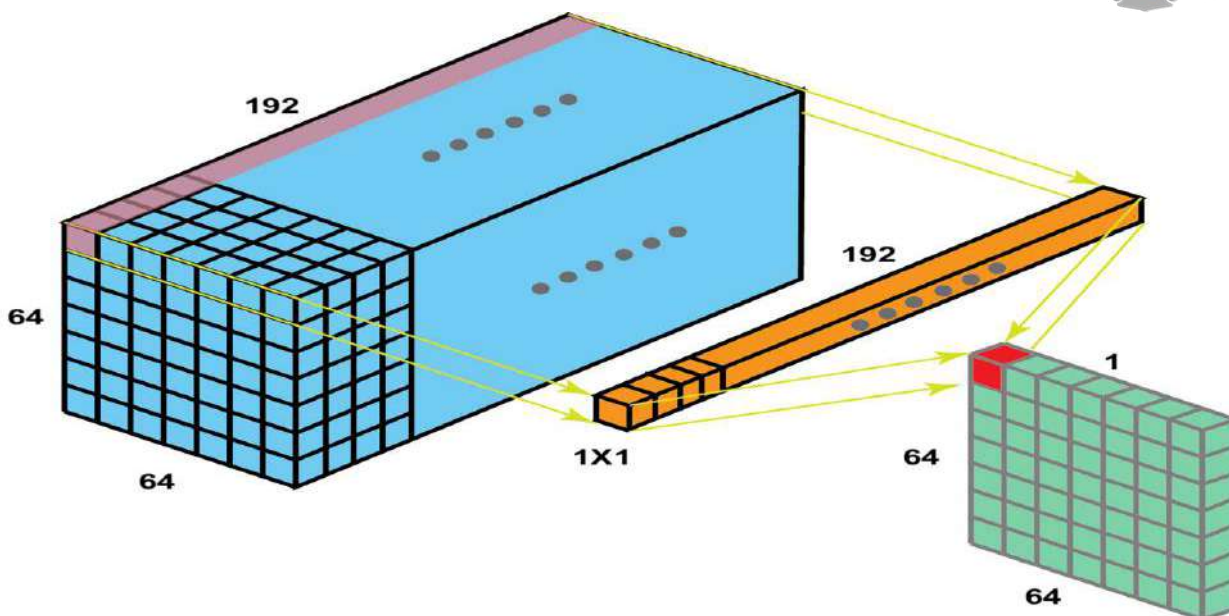


In Details (for more understanding)

1 x 1 conv was used to reduce the number of channels while introducing non-linearity.

In 1X1 Convolution simply means the filter is of size 1X1 (Yes — that means a single number as opposed to matrix like, say 3X3 filter). This 1X1 filter will convolve over the ENTIRE input image pixel by pixel.

Staying with our example input of 64X64X3, if we choose a 1X1 filter (which would be 1X1X3), then the output will have the same Height and Width as input but only one channel — 64X64X1. Now consider inputs with large number of channels — 192 for example. If we want to reduce the depth and but keep the Height X Width of the feature maps (Receptive field) the same, then we can choose 1X1 filters (remember Number of filters = Output Channels) to achieve this effect. This effect of cross channel down-sampling is called 'Dimensionality reduction'.

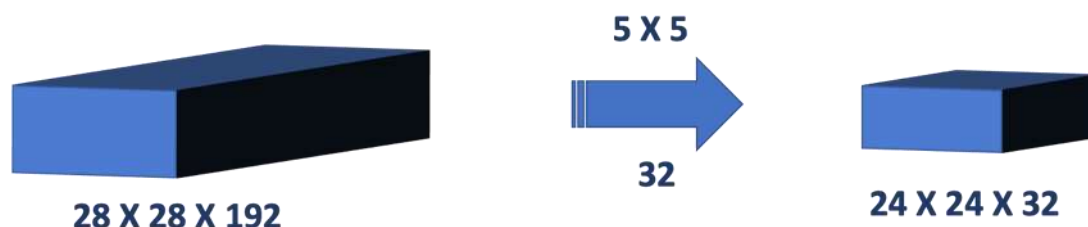


Now why would we want to something like that? For that we delve into usage of 1X1 Convolution

Usage 1: Dimensionality Reduction/Augmentation

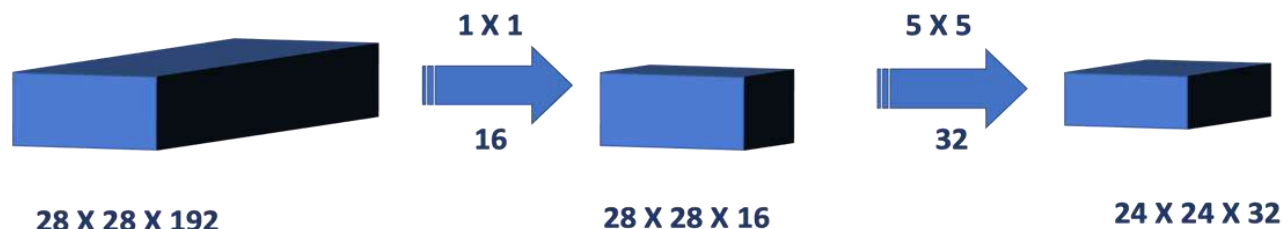
Winner of ILSVRC (ImageNet Large Scale Visual Recognition Competition) 2014, GoogleNet, used 1X1 convolution layer for dimension reduction “to compute reductions before the expensive 3×3 and 5×5 convolutions”

Let us look at an example to understand how reducing dimension will reduce computational load. Suppose we need to convolve 28 X 28 X 192 input feature maps with 5 X 5 X 32 filters. This will result in 120.422 Million operations



$$\text{Number of Operations} : (28 \times 28 \times 32) \times (5 \times 5 \times 192) = 120.422 \text{ Million Ops}$$

Let us do some math with the same input feature maps but with 1X1 Conv layer before the 5 X 5 conv layer



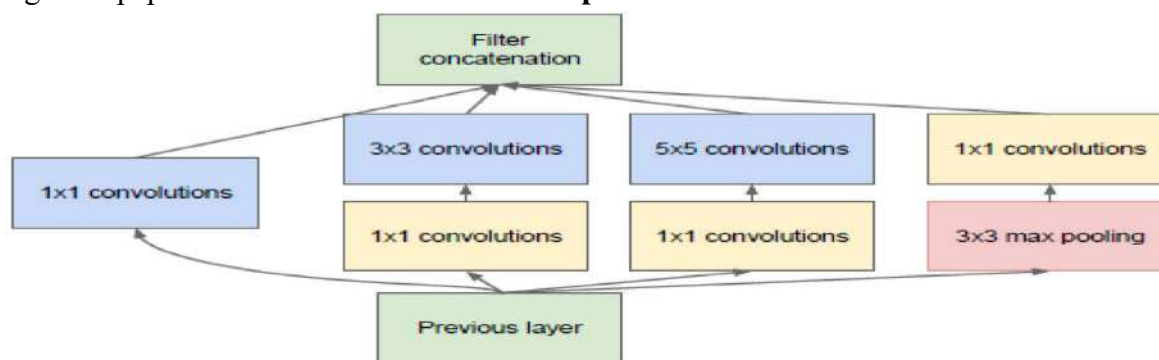
Number of Operations for 1 X 1 Conv Step : $(28 \times 28 \times 16) \times (1 \times 1 \times 192) = 2.4 \text{ Million Ops}$

Number of Operations for 5 X 5 Conv Step : $(28 \times 28 \times 32) \times (5 \times 5 \times 16) = 10 \text{ Million Ops}$

Total Number of Operations = 12.4 Million Ops

By adding 1X1 Conv layer before the 5X5 Conv, while keeping the height and width of the feature map, we have reduced the **number of operations by a factor of 10**. This will reduce the **computational needs and in turn will end up being more efficient**.

GoogleNet paper describes the module as “**Inception Module**”



(b) Inception module with dimensionality reduction

Usage 2: Building DEEPER Network (“Bottle-Neck” Layer)

2015 ILSVRC Classification winner, [ResNet](#), had least error rate and swept aside the competition by using very deep network using ‘Residual connections’ and ‘Bottle-neck Layer’.

In their paper, He et al explains (page 6) how a bottle neck layer designed **using a sequence of 3 convolutional layers with filters the size of 1X1, 3X3, followed by 1X1 respectively to reduce and restore dimension**. The down-sampling of the input happens in 1X1 layer thus funneling a smaller feature vectors (reduced number of parameters) for the 3X3 conv to work on. Immediately after that 1X1 layer restores the dimensions to match input dimension so identity shortcuts can be directly used. For details on identity shortcuts and skip connection, please see some of the Reviews on ResNet (Or you can wait for my future work!)



Usage 3: Smaller yet Accurate Model (“FIRE-MODULE” Layer)

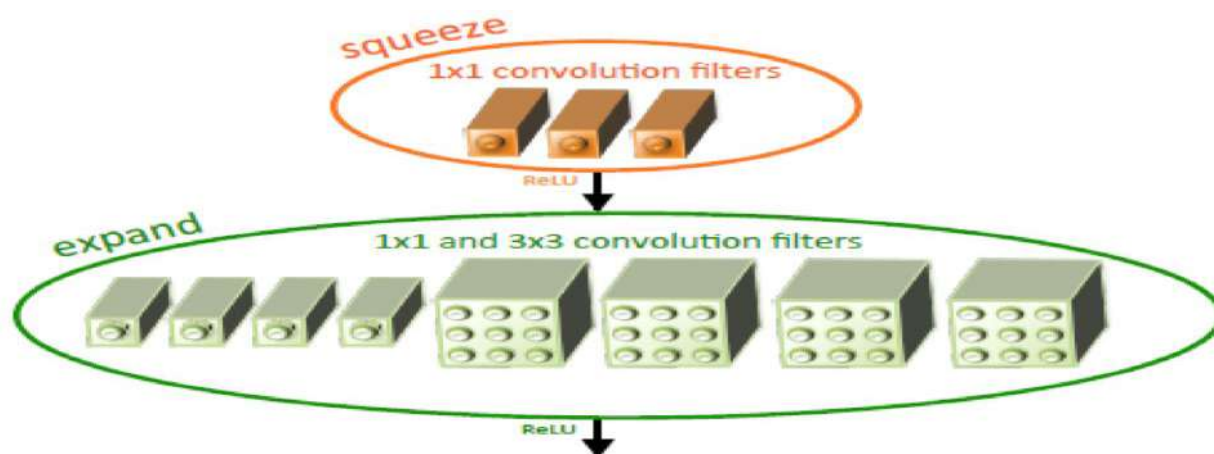
While Deep CNN Models have great accuracy, they have staggering number of parameters to deal with which increases the training time and most importantly need enterprise level computing power. Iandola et al proposed a CNN Model called **Squeeze Net** that retains AlexNet level accuracy while 50X times smaller in terms of parameters.

Smaller models have number of advantages, especially on use-cases that require edge computing capabilities like autonomous driving. Iandola et al achieved this by stacking a bunch of “**Fire Modules**” which comprise of

1. Squeeze Layer which has only 1X1 Conv filters
2. This feeds an Expansion layer which has mix of 1X1 and 3X3 filters
3. The number of filters in Squeeze Layer are set to be less than number of 1X1 filters + Number of 3X3 in Expand Layer

By now it is obvious what the 1X1 Conv filters in Squeeze Layer do — they reduce the number of parameters by ‘down-sampling’ the input channels before they are fed into the Expand layer.

The Expansion Layer has mix of 1X1 and 3X3 filters. The 1X1 filters, as you know, performs cross channel pooling — Combines channels, but cannot detect spatial structures (by virtue of working on individual pixels as opposed to a patch of input like larger filters). The 3X3 Convolution detects spatial structures. By combining these 2 different sized filters, the model becomes more expressive while operating on lesser parameters. Appropriate use of padding makes the output of 1X1 and 3X3 convolutions the same size so these can be stacked.

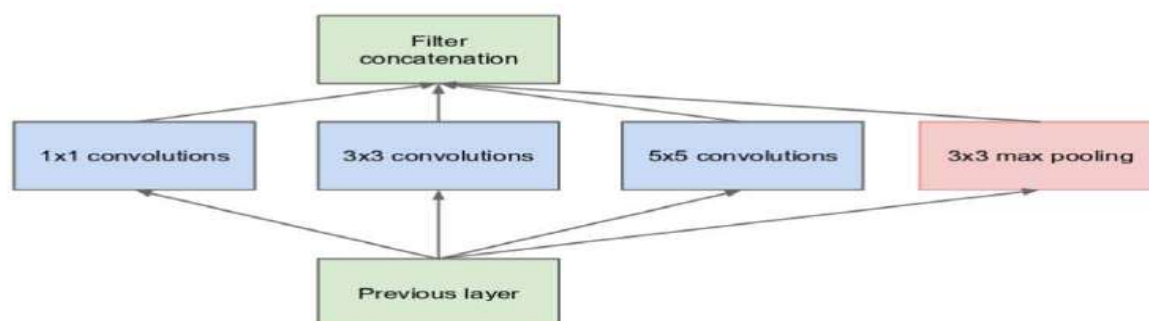


1X1 Convolution is effectively used for

1. Dimensionality Reduction/Augmentation
2. Reduce computational load by reducing parameter map
3. Add additional **non-linearity** to the network
4. Create deeper network through “Bottle-Neck” layer
5. Create smaller CNN network which retains higher degree of accuracy

10. What is an Inception Module?

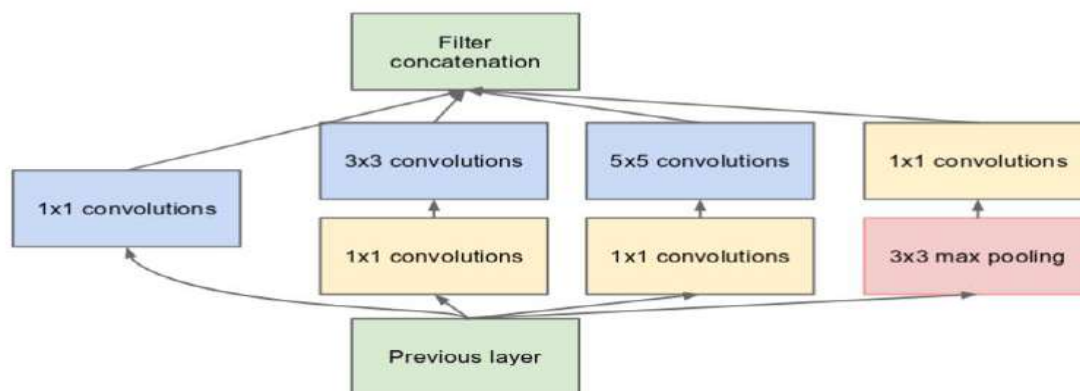
Inception Modules are used in Convolutional Neural Networks to allow for more efficient computation and deeper Networks through a dimensionality reduction with stacked 1x1 convolutions. The modules were designed to solve the problem of computational expense, as well as overfitting, among other issues. The solution, in short, is to take multiple kernel filter sizes within the CNN, and rather than stacking them sequentially, ordering them to operate on the same level.



(a) Inception module, naïve version

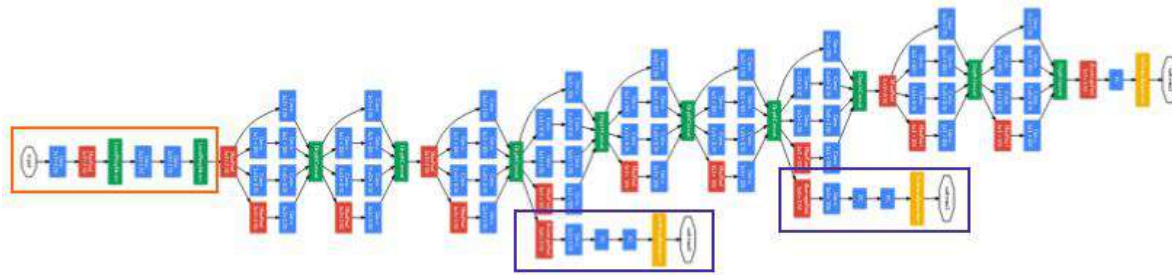
How does an Inception Module work?

Inception Modules are incorporated into convolutional neural networks (CNNs) as a way of reducing computational expense. As a neural net deals with a vast array of images, with wide variation in the featured image content, also known as the salient parts, they need to be designed appropriately. The most simplified version of an inception module works by performing a convolution on an input with not one, but three different sizes of filters (1x1, 3x3, 5x5). Also, max pooling is performed. Then, the resulting outputs are concatenated and sent to the next layer. By structuring the CNN to perform its convolutions on the same level, the network gets progressively wider, not deeper.



(b) Inception module with dimension reductions

To make the process even less computationally expensive, the neural network can be designed to add an extra 1x1 convolution before the 3x3 and 5x5 layers. By doing so, the number of input channels is limited and 1x1 convolutions are far cheaper than 5x5 convolutions. It is important to note, however, that the 1x1 convolution is added after the max-pooling layer, rather than before.



The design of this initial Inception Module is known commonly as GoogLeNet, or Inception v1. Additional variations to the inception module have been designed, reducing issues such as the vanishing gradient problem.

11. Input Channels

Imagine a network as a sequence of "layers", where each layer is of the form $x_{n+1} = f(x_n)$, where $f(x)$ is a linear transformation followed by a non-linearity such as sigmoid, tanh or relu. The layers operate on 3-D chunks of data, where the first two dimensions are (generally) the height and width of an image patch, and the third dimension is a number of such patches stacked over one another, which is also called the number of **channels** in the image volume. Thus, x can be viewed as a $H \times W \times C$ vector, where H, W are the dimensions of the image and C is the number of channels of the given image volume.

Multiple Input and Output channels

While we have described the multiple channels that comprise each image (e.g., color images have the standard RGB channels to indicate the amount of red, green and blue), until now, we simplified all of our numerical examples by working with just a single input and a single output channel. This has allowed us to think of our inputs, convolutional kernels, and outputs each as two-dimensional arrays. When we add channels into the mix, our inputs and hidden representations both become three-dimensional arrays. For example, each RGB input image has shape $3 \times h \times w$. We refer to this axis, with a size of 3, as the channel dimension.

12. Transfer learning

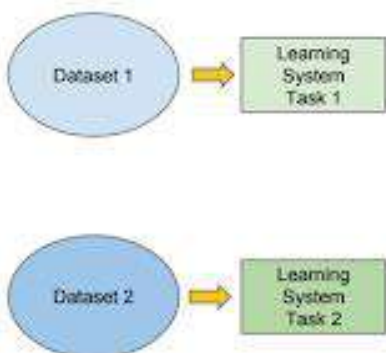
Transfer learning is the idea of overcoming the isolated learning paradigms and utilizing the knowledge acquired for one task to solve related ones, as applied to machine learning, and in particular, to the domain of deep learning.

Traditional ML

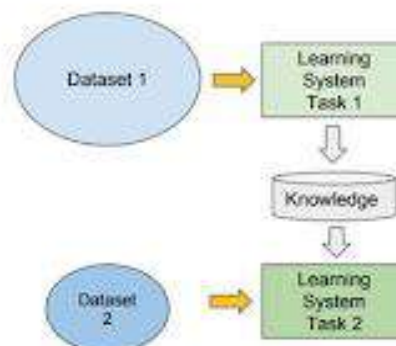
vs

Transfer Learning

- Isolated, single task learning:
 - Knowledge is not retained or accumulated. Learning is performed w.o. considering past learned knowledge in other tasks



- Learning of a new tasks relies on the previous learned tasks:
 - Learning process can be faster, more accurate and/or need less training data



Transfer Learning for Deep Learning Networks

Why transfer learning ?

Many deep neural networks trained on natural images exhibit a curious phenomenon in common: on the first layer they learn features similar to Gabor filters and color blobs. Such first-layer features appear not to be specific to a particular dataset or task but are general in that they are applicable to many datasets and tasks. As finding these standard features on the first layer seems to occur regardless of the exact cost function and natural image dataset, we call these first-layer features general. For example, in a network with an N-dimensional softmax output layer that has been successfully trained towards a supervised classification objective, each output unit will be specific to a particular class. We thus call the last-layer features specific.

In transfer learning we first train a base network on a base dataset and task, and then we repurpose the learned features, or transfer them, to a second target network to be trained on a target dataset and task. This process will tend to work if the features are general, that is, suitable to both base and target tasks, instead of being specific to the base task.

In practice, very few people train an entire Convolutional Network from scratch because it is relatively rare to have a dataset of sufficient size. Instead, it is common to pre-train a ConvNet on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the ConvNet either as an initialization or a fixed feature extractor for the task of interest. There is a myriad of strategies to follow for the transfer learning process in the deep

Therefore, a widely used strategy in transfer learning is to:

- Load the weights matrices of a pre-trained model except for the weights of the very last layers near the O/P,
- Hold those weights fixed, i.e. untrainable
- Attach new layers suitable for the task at hand, and train the model with new data

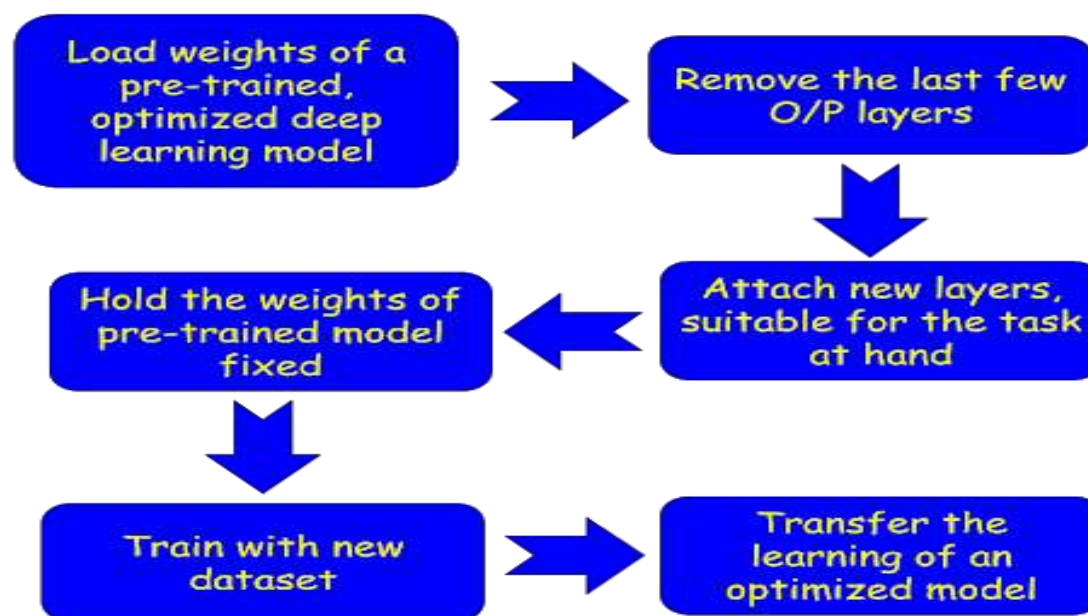


Fig: The transfer learning strategy for deep learning networks, as we will explore here.

This way, we don't have to train the whole model, we get to repurpose the model for our specific machine learning task, yet can leverage the learned structures and patterns of the data, contained in the fixed weights, which are loaded from the pre-trained, optimized model.

13. One shot learning

Deep Convolutional Neural Networks have become the state of the art methods for image classification tasks. However, one of the biggest limitations is they require a lots of labelled data. In many applications, collecting this much data is sometimes not feasible. One Shot Learning aims to solve this problem.

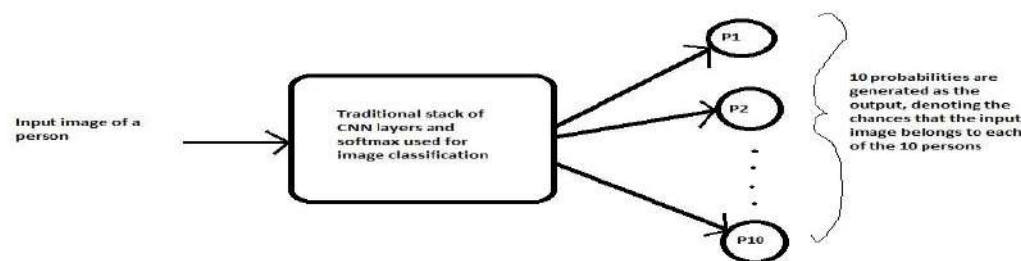
Classification vs One Shot Learning

In case of **standard classification**, the input image is fed into a series of layers, and finally at the output we generate a probability distribution over all the classes (typically using a Softmax). For example, if we are trying to classify an image as cat or dog or horse or elephant, then for every input image, we generate 4 probabilities, indicating the probability of the image belonging to each of the 4 classes. Two important points must be noticed here. **First**, during the training process, we require a **large** number of images for each of the class (cats, dogs, horses and elephants). **Second**, if the network is trained only on the above 4 classes of images, then we cannot expect to test it on any other class, example "zebra". If we want our model to classify the images of zebra as well,

then we need to first get a lot of zebra images and then we must **re-train** the model again. There are applications wherein we neither have enough data for each class and the total number classes is huge as well as dynamically changing. Thus, the cost of data collection and periodical re-training is too high.

On the other hand, in a **one shot classification**, we require only one training example for each class. Yes you got that right, just one. Hence the name **One Shot**. Let's try to understand with a real world practical example.

Assume that we want to build face recognition system for a small organization with only 10 employees (small numbers keep things simple). Using a traditional classification approach, we might come up with a system that looks as below:

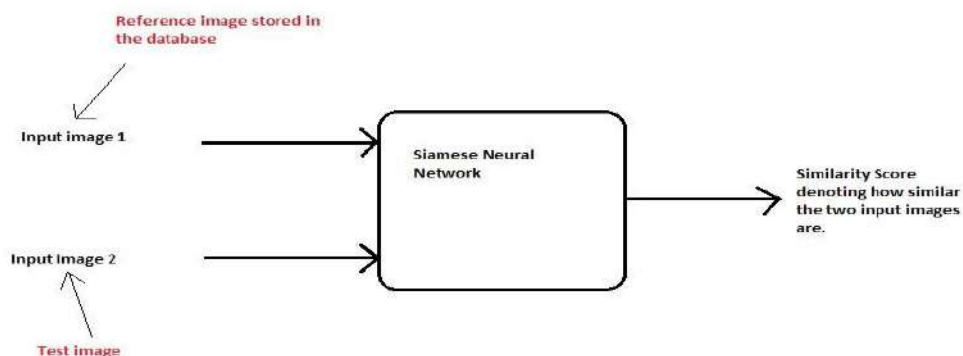


Problems:

a) To train such a system, we first require a lot of **different** images of each of the 10 persons in the organization which might not be feasible. (Imagine if you are doing this for an organization with thousands of employees).

b) What if a new person joins or leaves the organization? You need to take the pain of collecting data again and re-train the entire model again. This is practically not possible specially for large organizations where recruitment and attrition is happening almost every week.

Now let's understand how do we approach this problem using one shot classification which helps to solve both of the above issues:



Instead of directly classifying an input(test) image to one of the 10 people in the organization, this network instead takes an extra reference image of the person as input and will produce a similarity score denoting the chances that the two input images belong to the same person. Typically the similarity score is squished between 0 and 1 using a sigmoid function; wherein 0 denotes no similarity and 1 denotes full similarity. Any number between 0 and 1 is interpreted accordingly. Notice that this network is not learning to classify an image directly to any of the output classes. Rather, it is learning a **similarity function**, which takes two images as input and expresses how similar they are.

How does this solve the two problems we discussed above?

- a) In a short while we will see that to train this network, you do not require too many instances of a class and only few are enough to build a good model.
- b) But the biggest advantage is that , let's say in case of face recognition, we have a new employee who has joined the organization. Now in order for the network to detect his face, we only require a **single** image of his face which will be stored in the database. Using this as the reference image, the network will calculate the similarity for any new instance presented to it. Thus we say that network predicts the score in **one shot**.

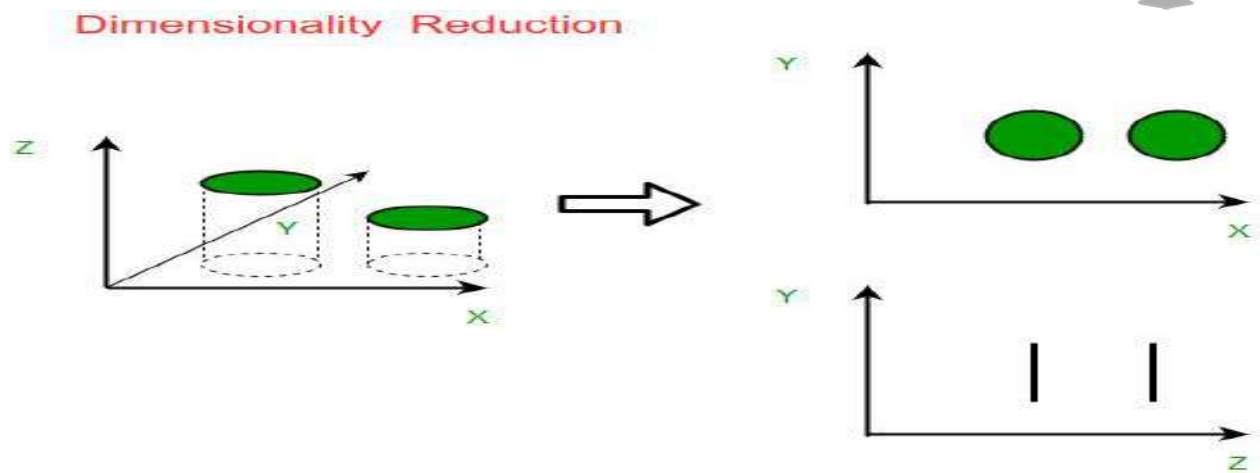
14. What is Dimensionality Reduction?

What is Dimensionality Reduction?

In machine learning classification problems, there are often too many factors on the basis of which the final classification is done. These factors are basically variables called features. The higher the number of features, the harder it gets to visualize the training set and then work on it. Sometimes, most of these features are correlated, and hence redundant. This is where dimensionality reduction algorithms come into play. Dimensionality reduction is the process of reducing the number of random variables under consideration, by obtaining a set of principal variables. It can be divided into feature selection and feature extraction.

Why is Dimensionality Reduction important in Machine Learning and Predictive Modeling?

An intuitive example of dimensionality reduction can be discussed through a simple e-mail classification problem, where we need to classify whether the e-mail is spam or not. This can involve a large number of features, such as whether or not the e-mail has a generic title, the content of the e-mail, whether the e-mail uses a template, etc. However, some of these features may overlap. In another condition, a classification problem that relies on both humidity and rainfall can be collapsed into just one underlying feature, since both of the aforementioned are correlated to a high degree. Hence, we can reduce the number of features in such problems. A 3-D classification problem can be hard to visualize, whereas a 2-D one can be mapped to a simple 2 dimensional space, and a 1-D problem to a simple line. The below figure illustrates this concept, where a 3-D feature space is split into two 1-D feature spaces, and later, if found to be correlated, the number of features can be reduced even further.



There are two components of dimensionality reduction:

- Feature selection: In this, we try to find a subset of the original set of variables, or features, to get a smaller subset which can be used to model the problem. It usually involves three ways:
 1. Filter
 2. Wrapper
 3. Embedded
- Feature extraction: This reduces the data in a high dimensional space to a lower dimension space, i.e. a space with lesser no. of dimensions.

Methods of Dimensionality Reduction

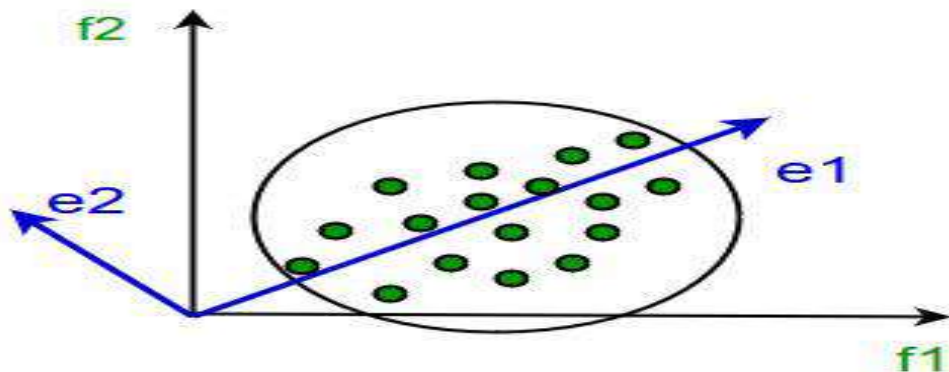
The various methods used for dimensionality reduction include:

- Principal Component Analysis (PCA)
- Linear Discriminant Analysis (LDA)
- Generalized Discriminant Analysis (GDA)

Dimensionality reduction may be both linear or non-linear, depending upon the method used. The prime linear method, called Principal Component Analysis, or PCA, is discussed below.

- **Principal Component Analysis**

This method was introduced by Karl Pearson. It works on a condition that while the data in a higher dimensional space is mapped to data in a lower dimension space, the variance of the data in the lower dimensional space should be maximum.



It involves the following steps:

- Construct the covariance matrix of the data.
- Compute the eigenvectors of this matrix.
- Eigenvectors corresponding to the largest eigenvalues are used to reconstruct a large fraction of variance of the original data.

Hence, we are left with a lesser number of eigenvectors, and there might have been some data loss in the process. But, the most important variances should be retained by the remaining eigenvectors.

Advantages of Dimensionality Reduction

- It helps in data compression, and hence reduced storage space.
- It reduces computation time.
- It also helps remove redundant features, if any.

Disadvantages of Dimensionality Reduction

- It may lead to some amount of data loss.
- PCA tends to find linear correlations between variables, which is sometimes undesirable.
- PCA fails in cases where mean and covariance are not enough to define datasets.
- We may not know how many principal components to keep- in practice, some thumb rules are applied.

- **Other methods to perform Dimension Reduction?(Optional)**

There are many methods to perform Dimension reduction. I have listed the most common methods below:

1. Missing Values: While exploring data, if we encounter missing values, what we do? Our first step should be to identify the reason then impute missing values/ drop variables using appropriate methods. But, what if we have too many missing values? Should we impute missing values or drop the variables?

I would prefer the latter, because it would not have lot more details about data set. Also, it would not help in improving the power of model. Next question, is there any threshold of missing values for dropping a variable? It varies from case to case. If the information contained in the variable is not that much, you can drop the variable if it has more than ~40-50% missing values.

2. Low Variance: Let's think of a scenario where we have a constant variable (all observations have same value, 5) in our data set. Do you think, it can improve the power of model? Ofcourse NOT, because it has zero variance. In case of high number of dimensions, we should drop variables having low variance compared to others because these variables will not explain the variation in target variables.

3. Decision Trees: It is one of my favorite techniques. It can be used as a ultimate solution to tackle multiple challenges like missing values, outliers and identifying significant variables. It worked well in our Data Hackathon also. Several data scientists used decision tree and it worked well for them.

4. Random Forest: Similar to decision tree is Random Forest. I would also recommend using the in-built feature importance provided by random forests to select a smaller subset of input features. Just be careful that random forests have a tendency to bias towards variables that have more no. of distinct values i.e. favor numeric variables over binary/categorical values.

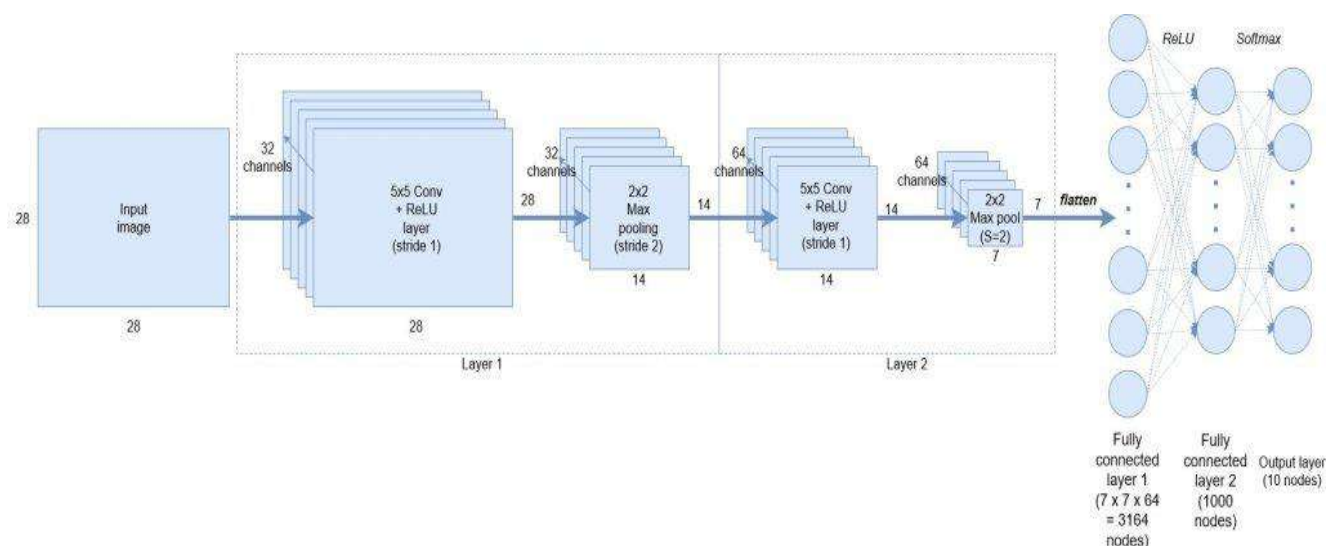
5. High Correlation: Dimensions exhibiting higher correlation can lower down the performance of model. Moreover, it is not good to have multiple variables of similar information or variation also known as "**Multicollinearity**". You can use *Pearson* (continuous variables) or *Polychoric* (discrete variables) correlation matrix to identify the variables with high correlation and select one of them using VIF (Variance Inflation Factor). Variables having higher value ($VIF > 5$) can be dropped.

15. Convolutional neural network?

Multi-layer neural networks can perform pretty well in predicting things like digits in the MNIST dataset. This is especially true if we apply some improvements. So why do we need any other architecture? Well, first off – the MNIST dataset is quite simple. The images are small (only 28 x 28 pixels), are single layered (i.e. greyscale, rather than a coloured 3 layer RGB image) and include pretty simple shapes (digits only, no other objects). Once we start trying to classify things in more complicated colour images, such as buses, cars, trains etc. , we run into problems with our accuracy. What do we do? Well, first, we can try to increase the number of layers in our neural network to make it *deeper*. That will increase the complexity of the network and allow us to model more complicated functions. However, it will come at a cost – the number of parameters (i.e. weights and biases) will rapidly increase. This makes the model more prone to overfitting and will prolong training times. In fact, learning such difficult problems can become intractable for normal neural networks. This leads us to a solution – convolutional neural networks.

- **A TensorFlow based convolutional neural network**

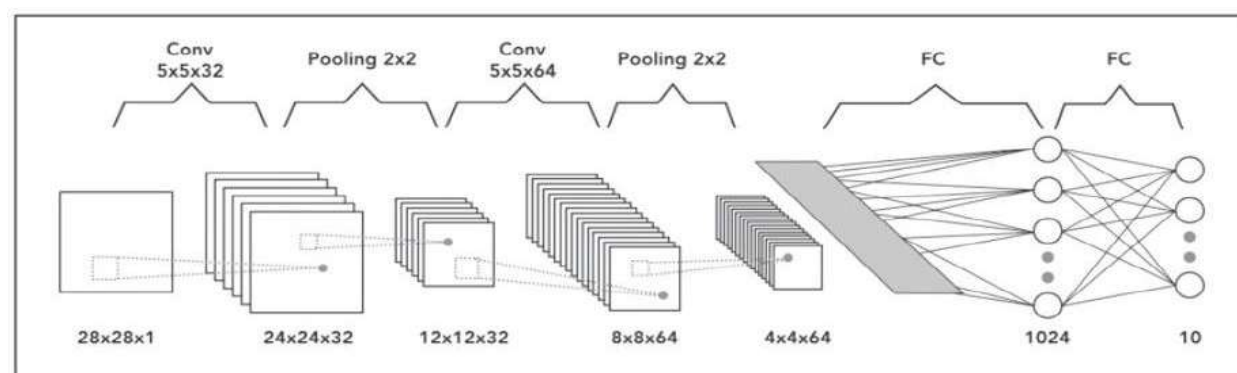
TensorFlow makes it easy to create convolutional neural networks once you understand some of the nuances of the framework's handling of them. we are going to create a convolutional neural network with the structure detailed in the image below. The network we are going to build will perform MNIST digit classification,



As can be observed, we start with the MNIST 28×28 greyscale images of digits. We then create 32, 5×5 convolutional filters / channels plus ReLU (rectified linear unit) node activations. After this, we still have a height and width of 28 nodes. We then perform down-sampling by applying a 2×2 max pooling operation with a stride of 2. Layer 2 consists of the same structure, but now with 64 filters / channels and another stride-2 max pooling down-sample. We then flatten the output to get a fully connected layer with 3136 nodes, followed by another hidden layer of 1000 nodes. These layers will use ReLU node activations. Finally, we use a softmax classification layer to output the 10 digit probabilities.

- **Keras**

3- Convolutional Neural Network Architecture



A Simple Example of CNN Architecture

First, we will define the Convolutional neural networks architecture as follows:

- 1- The first hidden layer is a convolutional layer called a Convolution2D. We will use 32 filters with size 5×5 each.
- 2- Then a Max pooling layer with a pool size of 2×2.
- 3- Another convolutional layer with 64 filters with size 5×5 each.
- 4- Then a Max pooling layer with a pool size of 2×2.
- 5- Then next is a Flatten layer that converts the 2D matrix data to a 1D vector before building the fully connected layers.
- 6- After that we will use a fully connected layer with 1024 neurons and relu activation function.
- 7- Then we will use a regularization layer called Dropout. It is configured to randomly exclude 20% of neurons in the layer in order to reduce overfitting.
- 8- Finally, the output layer which has 10 neurons for the 10 classes and a softmax activation function to output probability-like predictions for each class.

After deciding the above, we can set up a neural network model with a few lines of code as follows:

Step 1 – Create a model:

Keras first creates a new instance of a model object and then add layers to it one after the another. It is called a sequential model API. We can add layers to the neural network just by calling model.add and passing in the type of layer we want to add. Finally, we will compile the model with two important information, loss function, and cost optimization algorithm.

```
# Importing the required Keras modules containing model and layers

from keras.models import Sequential
from keras.layers import Dense, Conv2D, Dropout, Flatten, MaxPooling2D

# Creating a Sequential Model and adding the layers

model = Sequential()
model.add(Conv2D(32, kernel_size=(5,5), input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, kernel_size=(5,5), input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten()) # Flattening the 2D arrays for fully connected layers
model.add(Dense(1024, activation=tf.nn.relu))
```

```
model.add(Dropout(0.2))
model.add(Dense(10,activation=tf.nn.softmax))

#Compile the model

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Once we execute the above code, Keras will build a TensorFlow model behind the scenes.

Step 2 – Train the model:

We can train the model by calling `model.fit` and pass in the training data and the expected output. Keras will run the training process and print out the progress to the console. When training completes, it will report the final accuracy that was achieved with the training data.

```
model.fit(x=x_train,y=y_train, epochs=10)
```

Step 3 – Test the model:

We can test the model by calling `model.evaluate` and passing in the testing data set and the expected output.

```
test_error_rate = model.evaluate(x_test, y_test, verbose=0)
print("The mean squared error (MSE) for the test data set is: {}".format(test_error_rate))
```

Step 4 – Save and Load the model:

Once we reach the optimum results we can save the model using `model.save` and pass in the file name. This file will contain everything we need to use our model in another program.

```
model.save("trained_model.h5")
```

Your model will be saved in the Hierarchical Data Format (HDF) with `.h5` extension. It contains multidimensional arrays of scientific data.

We can load our previously trained model by calling the load model function and passing in a file name. Then we call the predict function and pass in the new data for predictions.

```
model = keras.models.load_model("trained_model.h5")
predictions = model.predict(new_data)
```

Summary

- We learned how to load the MNIST dataset and normalize it.
- We learned the implementation of CNN using Keras.
- We saw how to save the trained model and load it later for prediction.
- The accuracy is more than 98% which is way more what we achieved with the regular neural network.



RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at
<https://qp.rgpvnotes.in> .

If you have any queries or you want to submit your
study notes please write us at
rgpvnotes.in@gmail.com



LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in