Regularization

In the previous session, we have looked into the basic building blocks of neural networks. However, we need to understand the concept of regularization before proceeding to CNN's.

Regularization is a technique used to overcome the overfitting problem that neural networks suffer from during the training period. There are a few commonly used regularization techniques:

1. Squared Norm / Weight decay
2. Early Stopping
3. Data Augmentation
4. Dropout
5. Batch Normalization

- **Squared Norm / Weight decay:**

Due to the addition of the regularization term, the weights decrease because it assumes that a neural network with a smaller weight leads to a simpler model. Therefore, it will also reduce the overfitting problem to some extent.

The weight updation equation with the regularization term is given below. If the learning rate is high, the difference (**1-nl**) will be low and weights will be multiplied with a smaller quantity, hence becoming smaller

$$W_{i,j} - \frac{n\Delta(L)}{\Delta W_{i,j}} \qquad W_{i,j} * (1 - nI) - n\left[\frac{\Delta(L)}{\Delta W_{i,j}}\right]$$

Here,

$$n - learning\ rate$$

$$I - regularization\ parameter$$

$$(1 - nI) - weight\ decay$$

- **Early Stopping**:

Early stopping is a technique similar to cross-validation where a part of the training data is kept as the validation data. When the performance of the validation data starts worsening, we immediately stop the model's training.

We usually reduce overfitting by observing this training/validation accuracy gap during training, and then stop at the point this accuracy gap starts increasing.

In the figure below, the model training should be stopped at the dotted line, since after that the gap between training and testing error starts increasing, signifying that our model is overfitting on the training dataset.

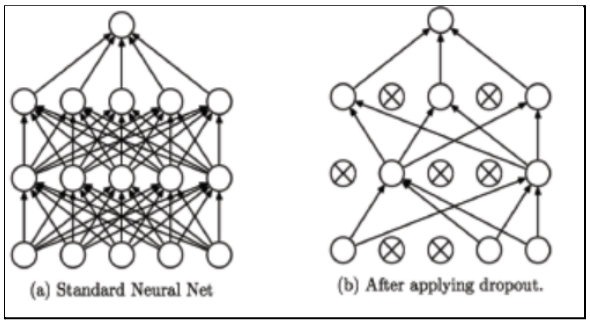
- **Data Augmentation**:

  The simplest way to reduce the overfitting problem is to increase the size of the training data. Data augmentation is a technique used to increase the quantity of the training data by adding slightly modified versions of the existing data. It acts as a regularization technique and solves the overall problem; this closely resembles the oversampling technique used in data analysis. This technique can provide a big leap in the accuracy of the model, since we are improving both the quantity and the diversity of the training data the model learns from.

- **Dropout**:

  A fully connected layer occupies most of the parameters, and hence, neurons develop co-dependency amongst each other during training which curbs the individual power of each neuron, leading to over-fitting of the training data.

  Dropout is another regularization technique that is used to solve the overfitting problem. It refers to the dropping of a few features and a few neurons in the hidden layer during the training phase. The dropping of features and neurons is chosen randomly by the neural network. The default dropout ratio is 0.5, at test time all the weights are multiplied by the chosen dropout ratio.



- **Batch Normalization**

  Batch normalization is a technique for improving the performance and stability of neural

networks. The idea is to normalize the inputs of each layer in such a way that they have a mean output activation of zero and a standard deviation of one.

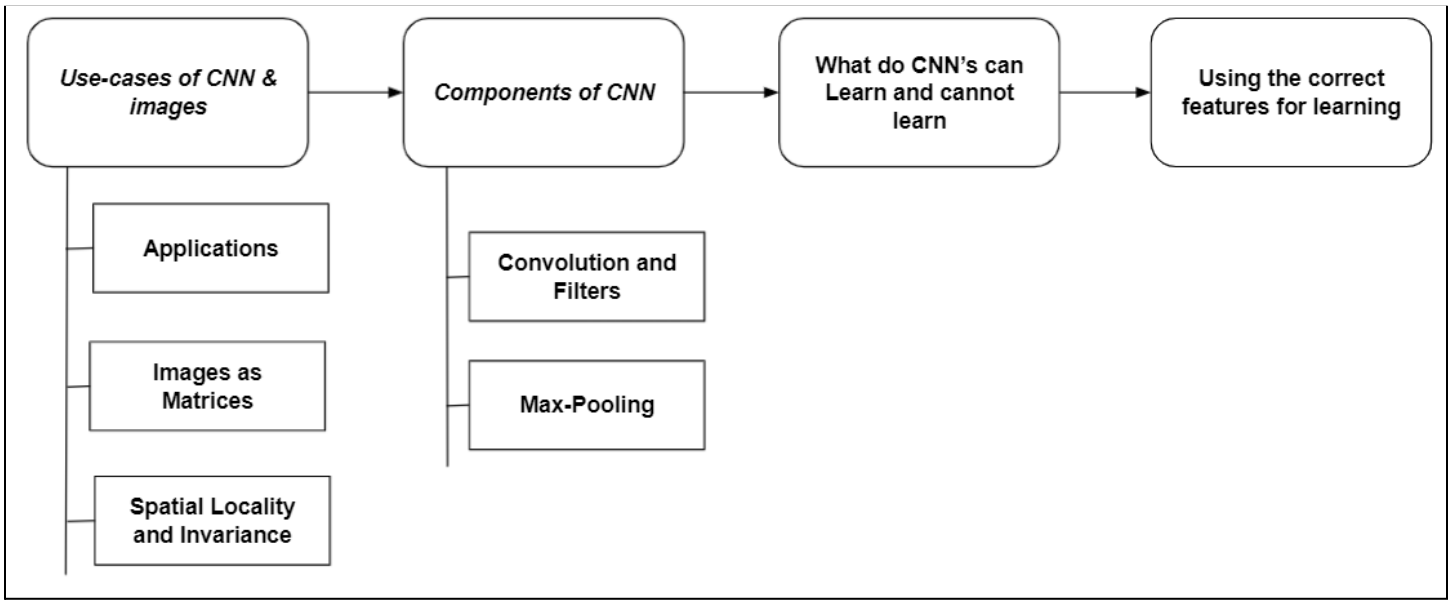
Due to these normalization “layers” between the fully connected layers, the range of input distribution of each layer stays the same, no matter the changes in the previous layer.

There are usually two stages in which Batch Normalization is applied:

1. Before the activation function / non-linearity
2. After the activation function / non-linearity

Now let us understand the concept of CNNs and why they are the preferred neural network architecture used to apply Deep Learning on images.

Deep Learning for Images (Convolutional neural network)



Use cases of CNN for Images (Applications)

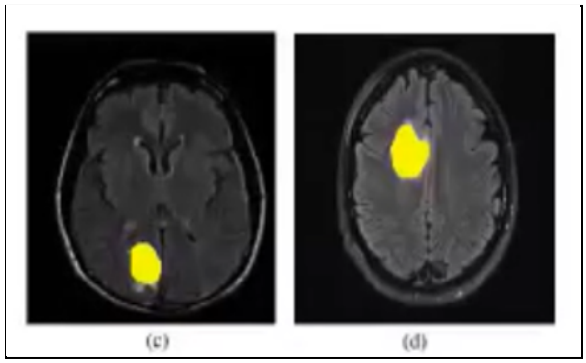
We have several use cases that can be resolved using CNNs, especially on image data. Some of them are:

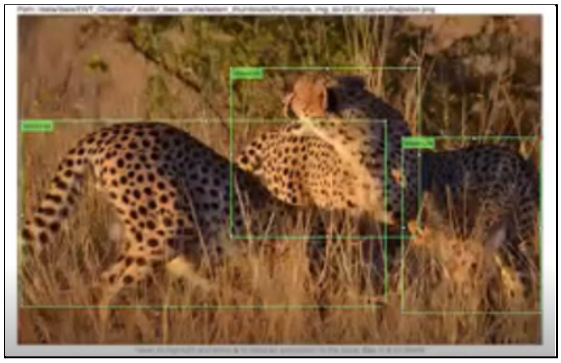
- **Autonomous Driving**: Here we mostly try to recognize the streets, cars, people, and obstacles in front of the car while driving so that we can automate the driving service of the car.
- **Medical Images**: Here, CNN can be used to recognize and predict tumors from the images of

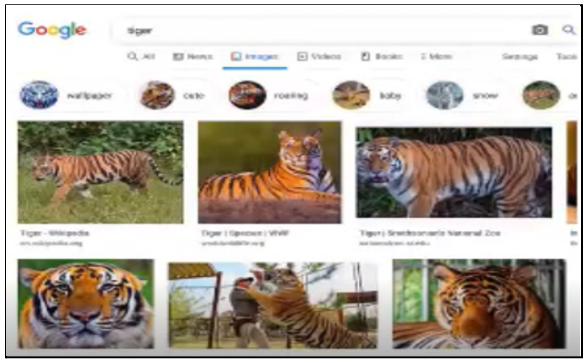
medical scans so that they can be treated properly.

- **Wildlife Monitoring**: We can detect and monitor animals in wildlife sanctuaries using our CNN models.
- **Image Search**: Our image search in Google is optimized by using deep learning models like CNNs for yielding quicker and better results.

There are many such use cases where applying CNN's on images can be used for object detection and recognition as well.



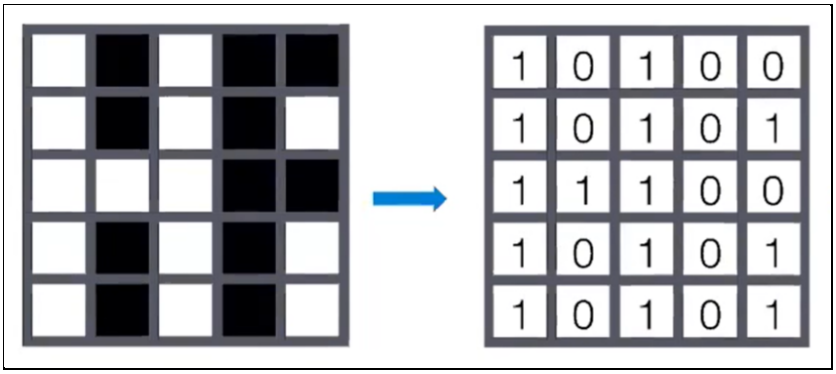




Images as Matrices (Tensors)

We have looked into several use cases or applications where we can use CNN's. The most important step to be followed is to extract the features from these images so that we can use these to train our models. Since all the Machine Learning and Deep Learning models work on numeric data, we need to convert these images into numerical values. And the most suitable mathematical representation for these images is matrices or *tensors*.

In Python, the images are converted into matrices in several ways, with the help of NumPy and OpenCV. But first, let us go through a simple mathematical representation of an image as a matrix:

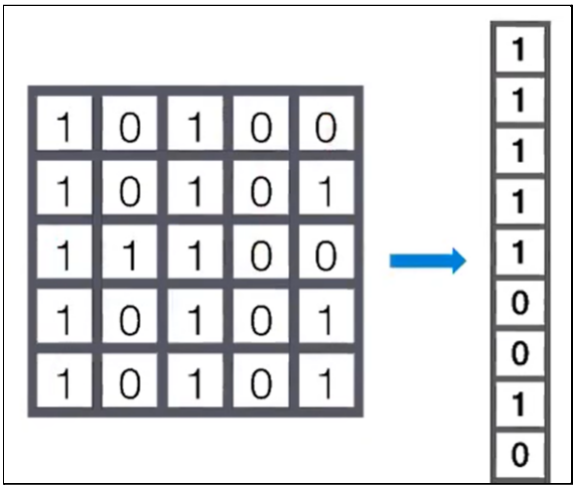


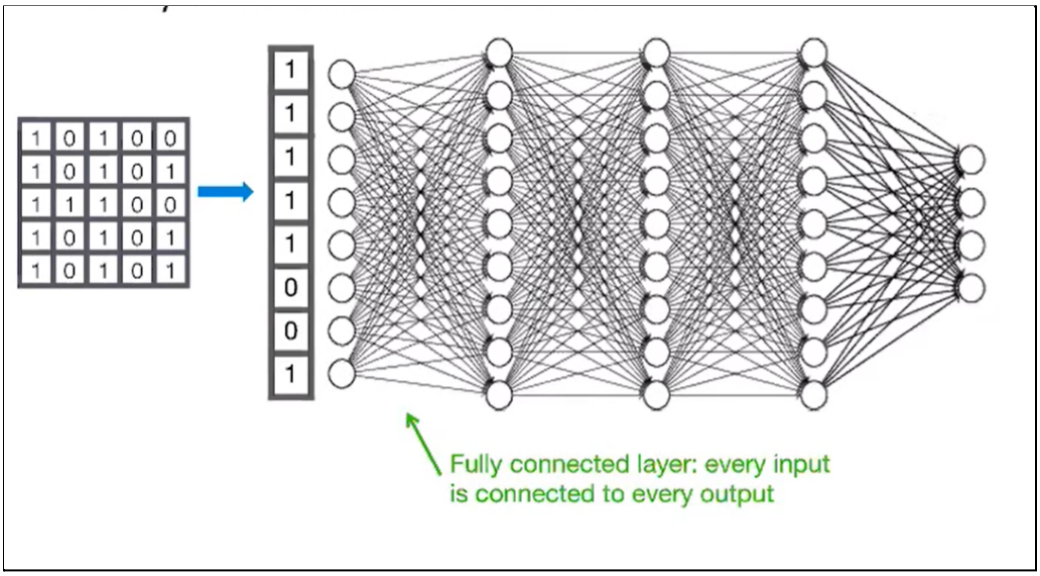
From the above image, you can observe that the brighter or white regions are represented by 1 and the darker regions are represented by 0. This is one of the simplest methods to represent an image as a matrix.

First, we will need to understand:

How are these images or matrices given as input to a neural network?

- In the image below, you will observe that the values in the matrix are taken column by column and stacked one over another vertically to form a 1d column vector. This vector is fed to the Fully Connected network.
- The stacking is not only limited to columns but it can also be done row by row and then used as a 1d column vector to be fed into the fully connected neural network.





- You will observe above how the matrix is converted into a 1d-vector and then fed into the Fully connected network. Here the important point to be considered is that each layer looks into the whole input vector and each neuron performs or works differently for the input.
- Now we will try to particularly think about all the methods that we can apply to know more about these images and extract the necessary information and features from these images during predictions.

Let us now go through an example to see:

What information do we actually need from images?

Let us take an example: **A cup detector**

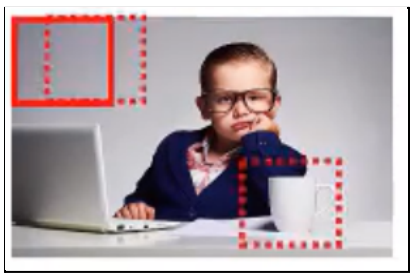
- From the given image, we will try to extract the important key points that would help us in detecting the cup in the image, So when we try to extract features, we may think that a cup always has an occurrence with a hand or a saucer, but that may sometimes not be true.
- In order to detect the cup correctly, we need to determine the local patch which has the highest chance for the occurrence of the cup. But this can be anywhere from the top corner to the bottom of the image. So there are different places where a cup may occur in an image.
- Hence, we want the cup to be detected similarly *wherever it may be* either at the bottom or at the top or to the left or right. This concept is called **Translational Invariance.**

Translational Invariance

- Invariance means we are blind towards something, and translational means shifting around. So Translational invariance means that we are blind to where exactly the cup may be found in the image. It does not make a difference no matter where the cup is in the image; being in a different location than expected should never be the reason a cup is classified as not a cup.
- Now we would like to observe whether all these features are reflected in our Fully Connected network or not. In a fully connected network, we just convert the whole image as a 1d-vector for input, so we are not preserving any 2d data at this point. However, we want to know if the cup is detected at small local patches in the image, meaning we should allow the neurons in the neural network to detect patterns in small local patches and detect the cup's presence in a specific location inside the image.
- That means we actually need to achieve two things here:
    - **Spatial locality**: Objects need to be detected in *local* patches.
    - **Translational invariance**: An object shifted anywhere inside an image still needs to be detected as that object.

The spatial locality requirement is achieved by using a **sliding window detector.**

- In the Sliding window detector, a small patch is taken into account and analyzed for the detection of the object. This patch shifts the window over the whole image until it finds the cup in the image.



So the basic idea for implementing the patch size detector is as follows:

- We will move in a hierarchy to learn a patch-size cup detector and slide it across the image.
- We will set a trigger flag as “yes” if it fires or the object is detected anywhere.
- Our hierarchy will be in the following order: first we will detect edges, then shapes, then parts, objects, and finally scenes.

  Hierarchy: edges -> shapes/textures -> parts -> objects -> scenes.

When learning the cup detector, we obviously want that the detector always detects a cup in the image, rather than a hand or a human. This is taken care of with the help of the loss function and minimizing this loss function for cup detection. This necessitates the use of data augmentation that we discussed in the previous lecture.

Data Augmentation would add some variances and generate new images either zoomed in or out, tilted, rotated, etc so that our detector can detect the cup in the image with any variance present. Even the size of an image is an important point to be considered, as the network should learn that cups can come in different sizes as well.

Let us look at how all of this is actually achieved by a neural network:

- **Convolution:** Convolutions help us in finding the spatial locality through local detectors that are learning a patch size cup detector, by sliding them across the image.

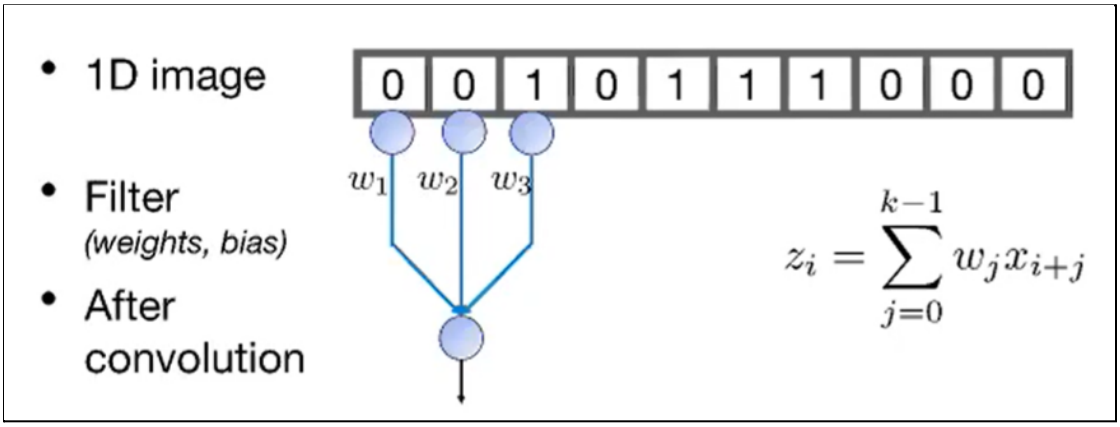
- **Weight Sharing**: We need to apply the same detector to all the image patches to achieve translational invariance and make the detector work efficiently.

- **Pooling**: This will abstract away the locality and just trigger “yes” if a cup is present anywhere in the region.

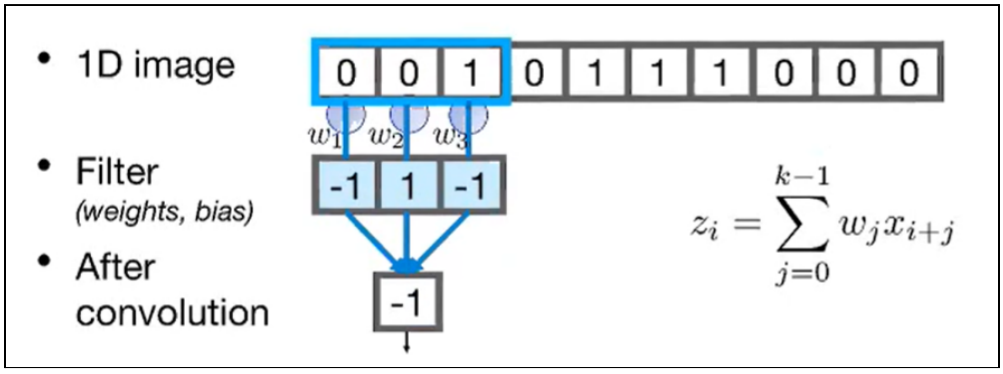
Convolution

1D Example for Convolution

Let us look at an example to understand convolution better.
We will use a 1D-array as an input to understand the convolution operation. Here we will have a patch size of 3 so that we can use our filter on 3 cells at a time.

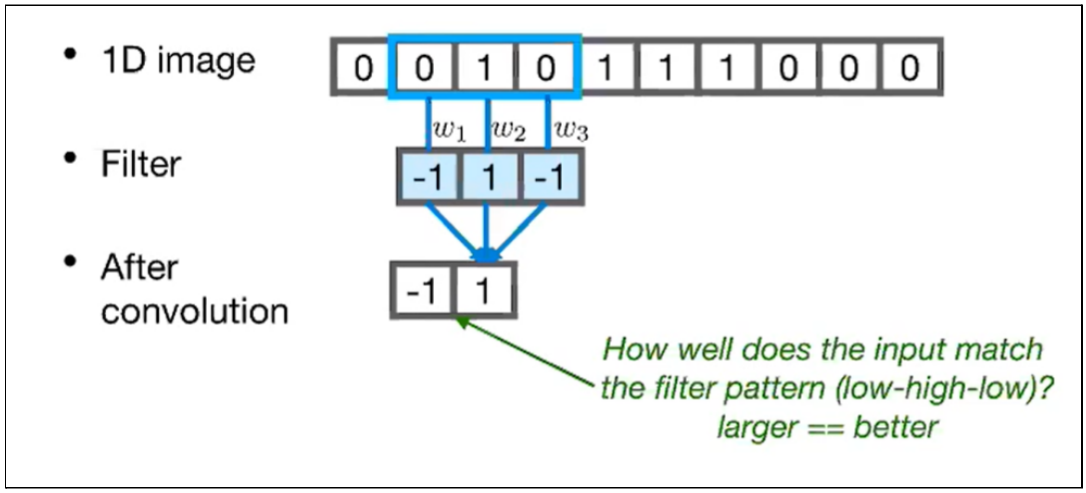


- In the above image, we observe a 1D array with inputs 0 and 1 where 1 would represent the brighter region and 0 would represent the darker region of the image. Here we also have a filter with weights w1, w2, w3, and the bias b has been initialized with 0.
- Now let us take the weighted combination of the inputs with the respective weights. A filter is a small unit that looks at a local patch and takes the weighted combination of the inputs and weights. Thus the filter does the pattern matching, because as is apparent here, matching patterns between inputs and weights give higher final scores.



- So by taking the weighted combination of inputs and weights for the first patch we get the output as -1. This is because: -1*0 + 1*0 + -1*1 = -1.

- The output tells us how well the input matches the filter pattern. Here in our example, our filter pattern is low, high, low whereas the input pattern is low, low, high - so it gives a low output due to the mismatch in inputs & weights.
- Now we slide the filter by a shift of 1 and again calculate the weighted sum, and this process goes on until the end of the array. This is the concept of **Weight Sharing**.
- We share the same weight of the filter with each patch and we calculate the weighted sum. This is markedly different from what was being done in the fully connected neural network.

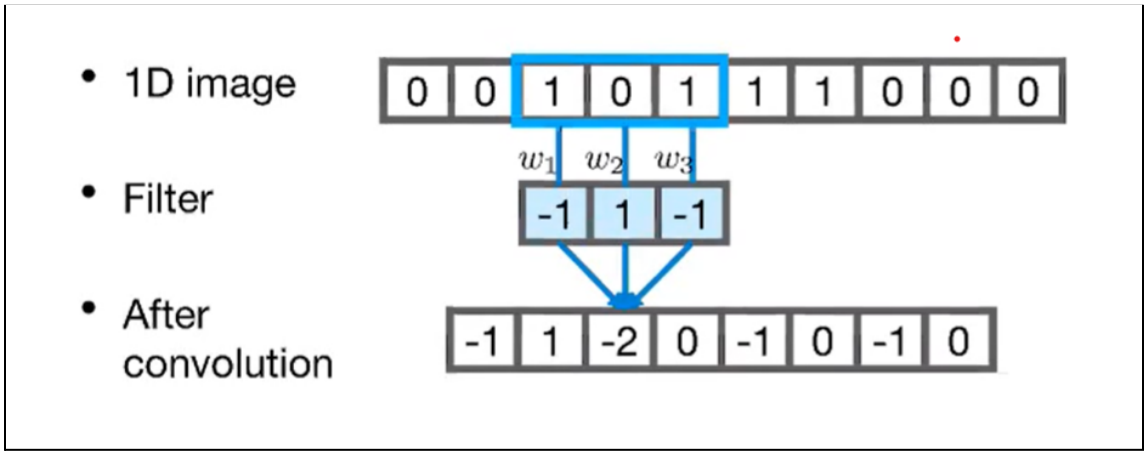


- Here we have just used a sample filter of low, high, low but this can be any filter. There are many other filters used in CNNs, we will look into them later.
- The number of cells we shift the filter by is known as **Stride**.

Stride

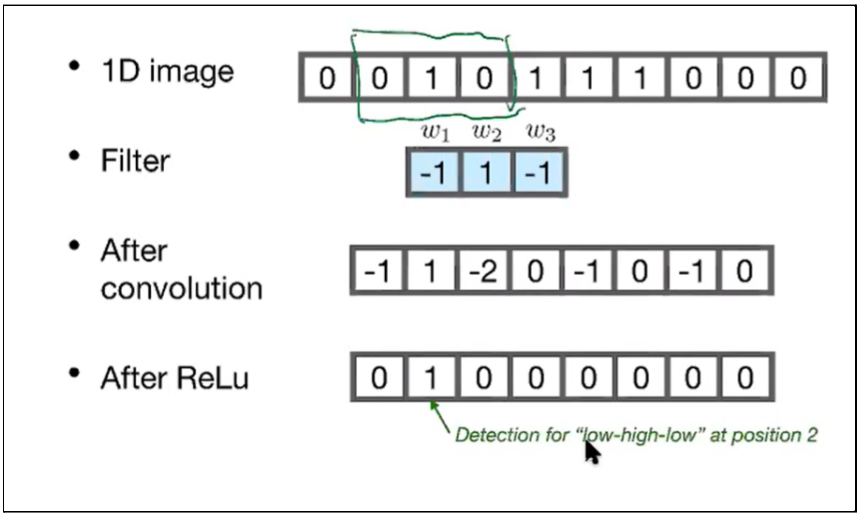
- When performing convolution, we observe that we slide from one corner to another corner through a shift at each step; this shift is called stride.
- If we have stride = 1, then it slides from one window to the other by a shift of 1.
- Thus stride helps us in dimensionality reduction, making the output dimension an integer rather than a fraction and we can also observe that as the number of strides increases the computational power required for computation decreases, since the size of the output decreases as well.

So due to weight sharing, a CNN requires far fewer weights in comparison to fully connected neural networks.



After we have performed the convolution on the input data using a filter, the other important operation is applying the non-linear functions like Tanh, Sigmoid, and ReLu on the output of the filter here. Here we will use ReLu as a non-linear activation function for our example.

ReLu(x) is simply Max(0,x): i.e. values below 0 become 0, and other values above 0 are kept as it is.

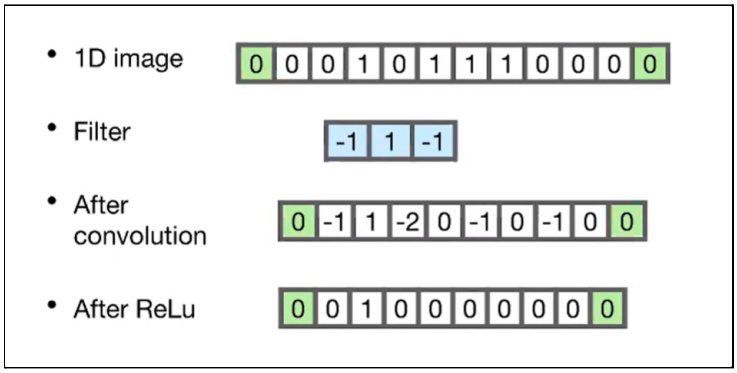


We observe that after the ReLu has been applied, we get 1 at position 2 whereas all the other positions become zeros.

- One important observation here is that the output is smaller than the input as the cells in the output are lesser in number. This is happening because of our filter. The size of our filter is 3, which means the number of times we can fit our filter on the input is less than the number of cells in the input. So the output typically gets shrunk. The solution to this problem is **Padding**.

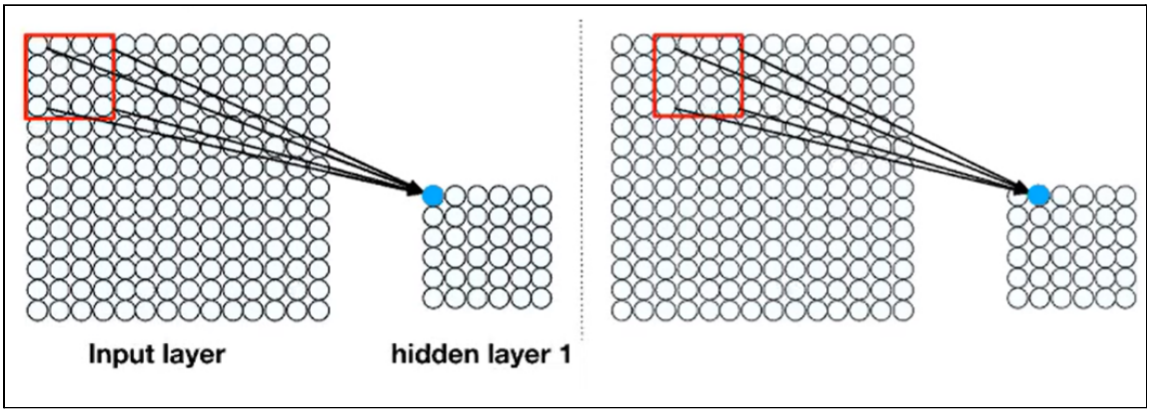
Padding

- Padding is a simple trick to add extra pixels to the edges or boundaries of the image so that the effective size of the image gets increased and there is no loss in the original number of pixels from the image. The boundary edges are preserved during convolution.
- One of the basic types of padding is called Zero Padding. In this, we add zeros to all the edges of an image to prevent the loss of the original number of pixel values of the image.
- We usually perform zero padding on the corners of the tensor. We do this to get a size of output that is the same as that of the input.
- Here in the below image, we observe that we have padded both sides of the tensor with Zeros.

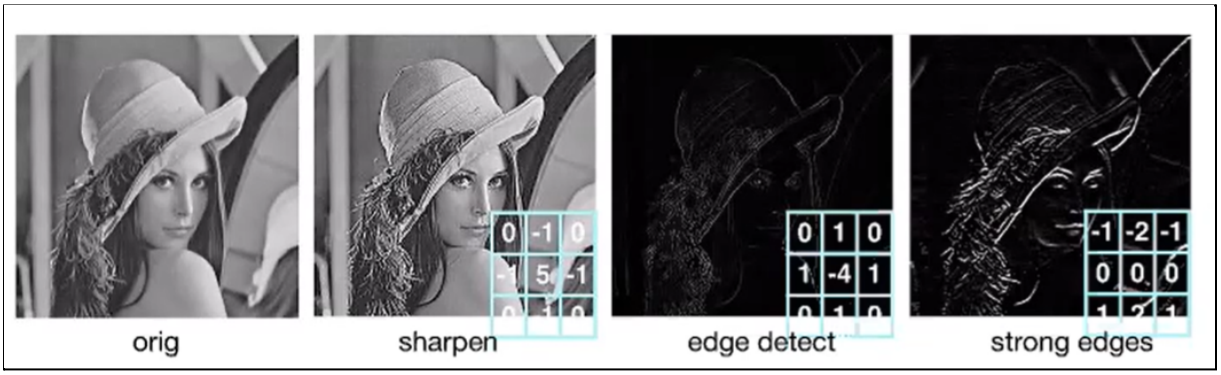


2D Convolution

- In 2D convolutions, we do both, we shift right as well as down for convolution, thus here we shift in two directions.
- Let us look into an example of 2d-convolution, here we use a 4x4 filter with stride=2. So here we will shift by a slide of 2 to the right and after the row is completed, we will slide down the matrix and complete the next row, and so on until all the rows of the new output matrix are obtained.



- Let us look at an example to see how different filters give us different outputs and are used differently.



- All the three different 3x3 filters behaved differently on the original image owing to the difference in their weights.
- Some of these filters sharpen the image, some detect the edges and some add intensity to the edges.
- Similarly, there are other filters like the Sobel filter, Prewitt filter, and the Laplacian filter. Let us briefly look at them.

Prewitt filter

It is mostly used to detect horizontal and vertical edges in an image.
This filter has three main properties:

- Some of the masked elements in the filter should be equal to zero.
- The elements present in the filter should have opposite signs.
- If there are more weights then there are higher chances of edge detection.

It provides us two masks for detecting edges both in horizontal and vertical directions.

-1	-1	-1
0	0	0
1	1	1

-1	0	1
-1	0	1
-1	0	1

Sobel filter

- These filters are similar to the Prewitt filters with only some value changes in the filter.
- Here the center values are marked as +2 and -2 instead of +1 and -1.
- This gives a better intensity at the edge regions in comparison to the Prewitt filter.

1	2	1
0	0	0
-1	-2	-1

1	0	-1
2	0	-2
1	0	-1

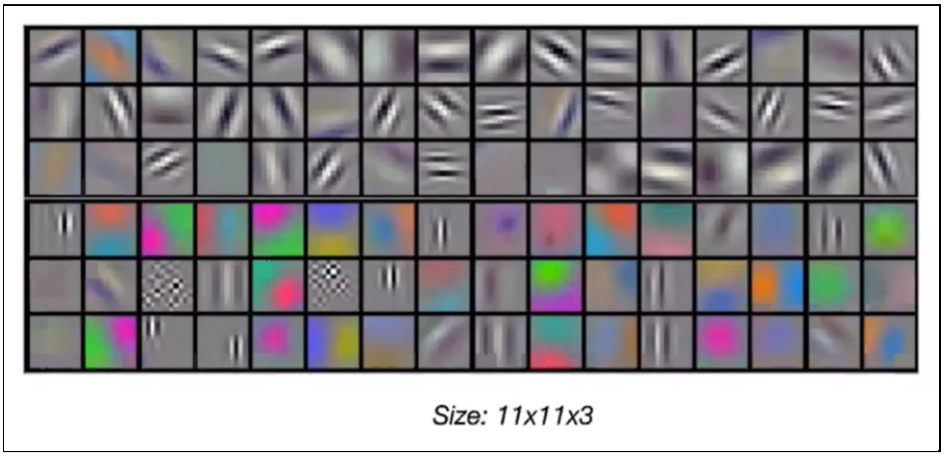
Laplacian filter

- It is also an edge detector filter, used for detecting edges by computing the second derivatives of an image. It measures the rate of change of the first derivative of the image.
- The Laplacian filter behaves differently by highlighting the regions where pixel intensities change abruptly. Thus it detects edges in an image.
- The array given below is the 3X3 Laplacian filter.

0	-1	0
-1	4	-1
0	-1	0

Laplacian filter

Let us look at an example of the filters actually learned by a network:

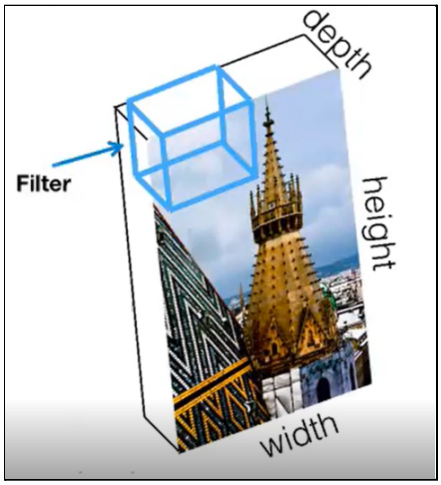


We can observe that some of these filters detect colors, and others detect the horizontal or vertical edges. This is actually a 3d filter, since we have color in these filters (the third dimension is color channel - R, G and B). As we've seen before, due to these filters/weights being shared across the layer, we have far fewer parameters here in comparison to Fully connected neural networks.

Similar to 2d convolution, let us now look into the idea of 3d convolutions.

3D Convolution

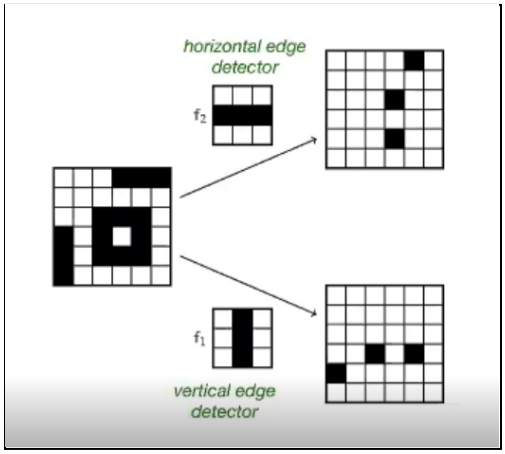
- The term 3d convolution simply refers to color images, as color is the third dimension in an image. So images are just 3d-tensors or matrices.
- Due to the 3d nature of the image, the filter needs to be a 3d Tensor here as well. So in addition to height and width, the filter will also have depth.



Over and above simply detecting edges however, there is a necessity for us to detect shapes (combinations of these edges) as well. This is the reason we use **multiple filters or a filter bank.**

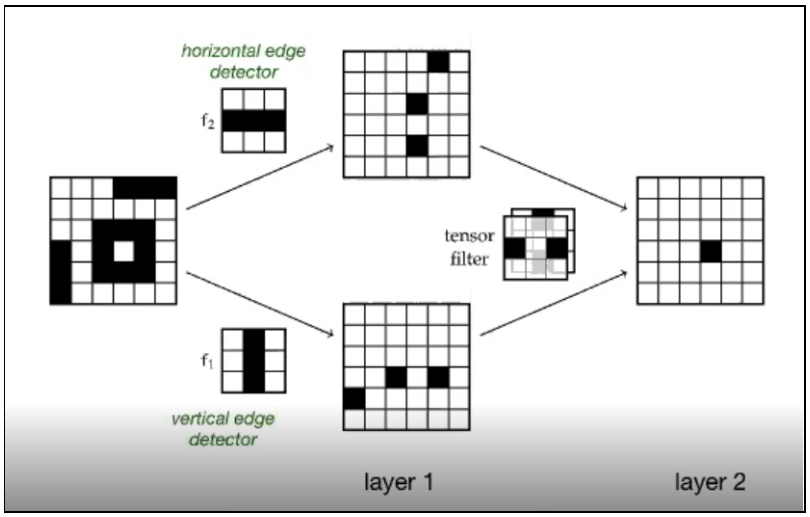
Multiple Channels/Filters

- As we need to detect shapes using filters, we need a specific combination of multiple types of edges to detect each kind of shape.



- Here in this image, we have both horizontal and vertical edge detectors; one detects the horizontal edges whereas the other detects vertical edges. If we need to detect a square box that has both horizontal and vertical edges, we need to use both these detectors to detect both the horizontal and vertical edges and combine these to get an output which can confirm the presence of a square box.

- In the first layer we use both the detectors to detect horizontal and vertical edges, but in the second layer we combine both of these to get a final output detected.

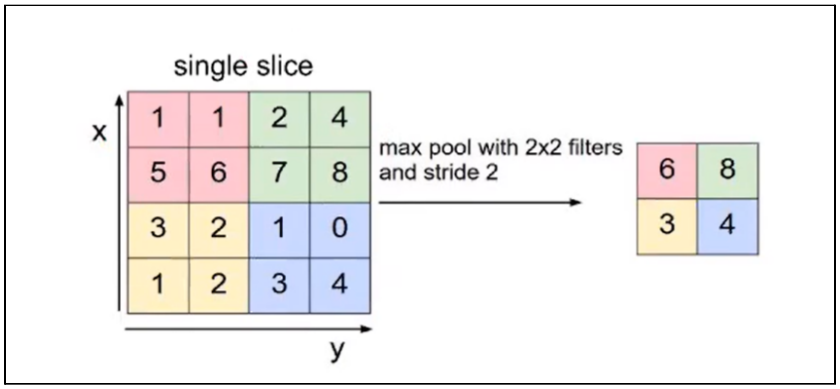


- Finally in layer 2, we get a “yes“ trigger (black cell) only when a box actually gets detected.

- Until now, we have looked into how the convolution occurs and edges are detected by sliding over each patch in an image. Now we need to establish the use of spatial locality and translational invariance, which will tell us that the object or cup can be detected anywhere, and this is done using another layer called **Max Pooling.**

Max Pooling

- Max Pooling is similar to our filter method. However, it does not take the weighted sum of the patch but merely the maximum element in the patch. Also, it has no learnable parameters as it only takes the maximum from the given values.
- Max Pooling has two advantages:
    - It abstracts away the locality so that the location does not matter, as the object can be found anywhere in the image.
    - It reduces the dimension or size of the output / processed image.



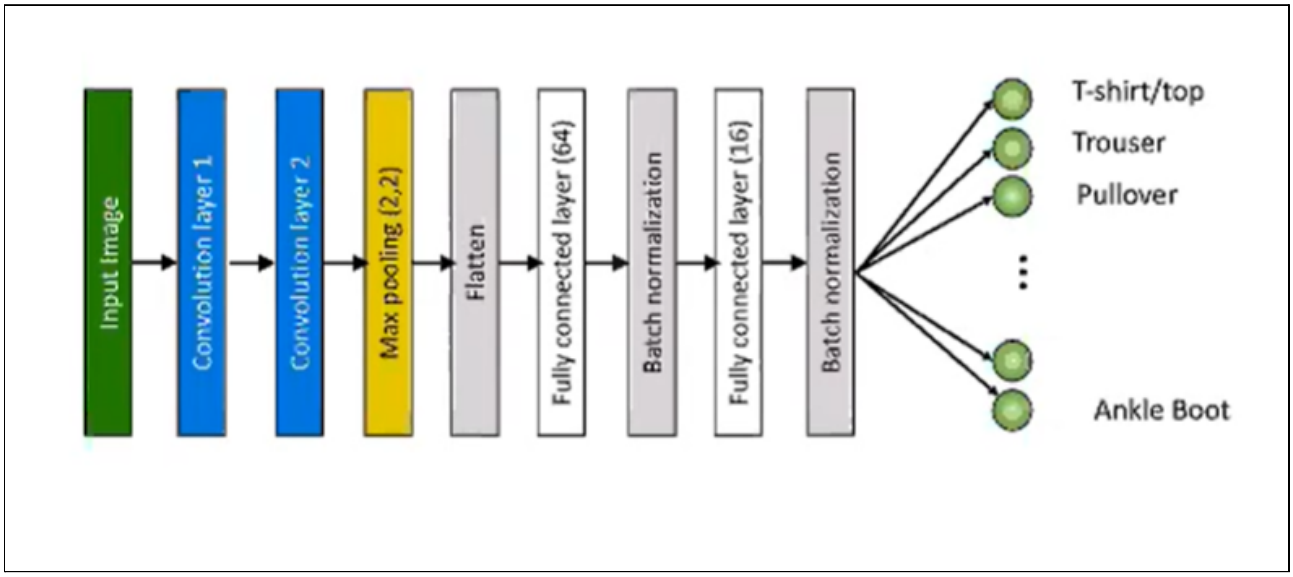
- Here we observe that the filter has a size of 2x2 with stride=2, so for max-pooling, we take the max of the first 2x2 block elements colored in red (6), and similarly, we do that for other blocks to get the maximum values.
- Max pooling is usually applied per channel, and not across the channel.

Now that we have learned about all the individual layers, let us look briefly into a case study to see how the CNN architecture performs on it.

Case Study: CNN for Fashion MNIST

- In the previous session, we have used a fully connected network on the Fashion MNIST dataset, so we will now try to use CNN on the same dataset.

- We will use the below-given architecture for building our CNN model:

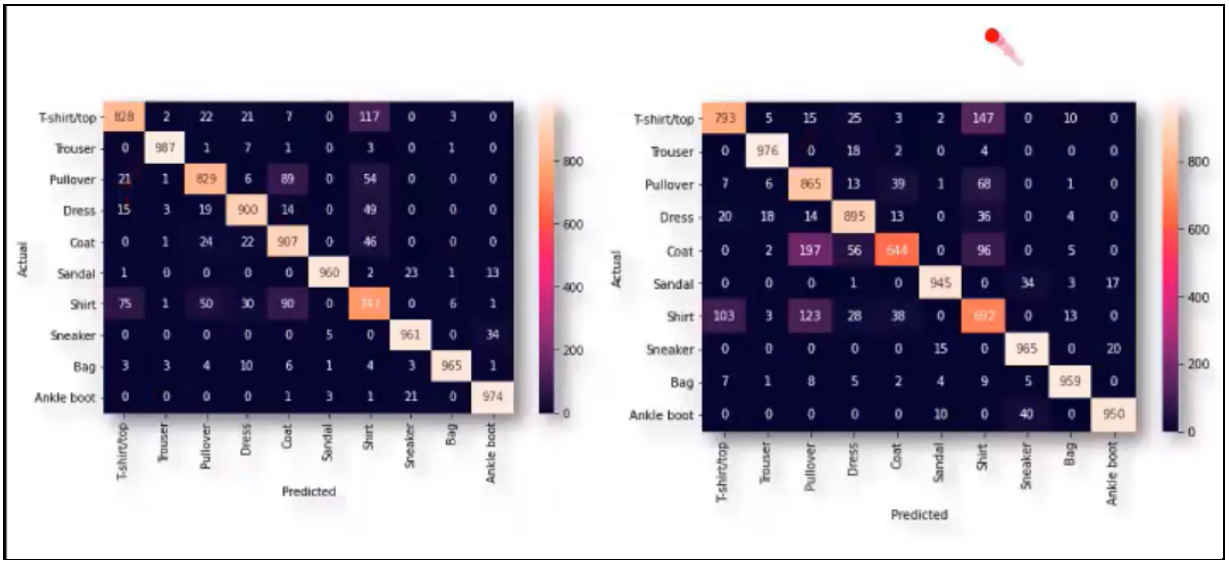


- In this architecture, we have 2 convolution layers with 16 filters of size 3x3.
- Then we perform the max pooling with size 2x2.
- The first three layers, which are 2 convolutions and 1 max-pooling, are best used for learning the **representation** of the image.

- Then we flatten the output after max pooling, and use 2 fully-connected layers, each having its own Batch Normalization layer. These 4 layers are especially used for prediction, so these layers can be termed **classifiers**.
- The architecture has been designed such that we first learn the patterns in the image using the representation layers, and then we use the fully-connected layers to act as a classifier.
- The flatten layer is required so that the output of the convolution layers, which is also an image or a matrix, is flattened into one dimension so that it can be fed into the fully connected network.
- We also use Batch Normalization with the fully connected layer in order to stabilize the output.
- Finally, we use a softmax activation function to output probabilities of 10 different classification categories.
- The number of layers here can be increased or decreased depending on the computational resources available and the amount of data we have.
- So we can try to alter the number of layers and change the architecture to check if we could have made the model better by improving its accuracy.

- Now we train the model using the Adam optimizer for about 10 epochs with a mini-batch size of 128. Adam usually utilizes squared gradients to scale the gradient and behaves similarly to SGD with Momentum, to learn about the moving average of the gradient.
- Our mini-batch would look at 128 data points at one time and compute the gradient for all of them, then improve the average performance and take one step, and then it looks at the next 128 data points, and so on.
- When comparing the accuracy of the models, we see that this CNN model has 91% accuracy, which is an improvement on the earlier ANN (fully-connected) which only had 87% accuracy.

Confusion Matrix: CNN vs Fully Connected

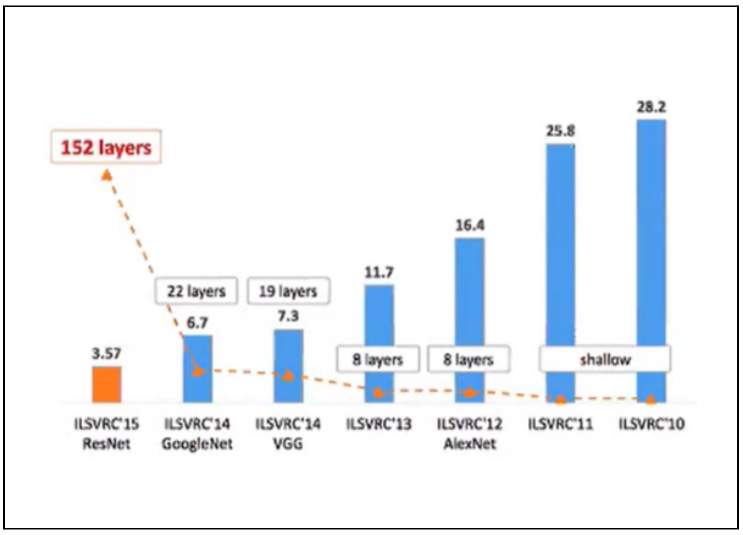


- In the image above, the confusion matrix on the left side is for the CNN, and the one on the right is for the fully connected neural network.
- If you observe the right side image, you will see that the ANN was confused about a few classes (confused a coat for a pullover, confused a t-shirt for a shirt), whereas the CNN on the left performed better in those classes.
- So we can say that the CNN is a better pattern detector for images than the ANN.

Let us see what happens when the neural network becomes deeper and deeper.

How deep can CNNs be?

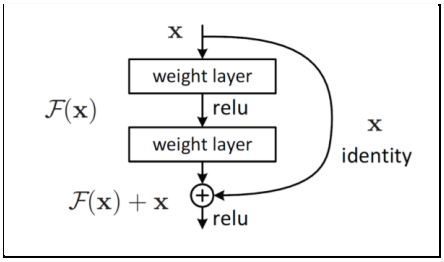
- Let us answer this question by using the famous ImageNet challenge, which has millions of images and has acted as a benchmark dataset to measure the performance of CNNs.



- We observe in the image above that as the number of layers has increased in recent times (right to left), the performance of the model has also increased (smaller loss on the bar graph).
- However, neural networks run into a new set of problems when they become deeper:
    - The number of parameters in the network drastically increases.
    - The problem of vanishing gradients starts to become severe due to the chained product of various gradients in the whole network.
    - Deeper neural networks may also overfit heavily on the training dataset.

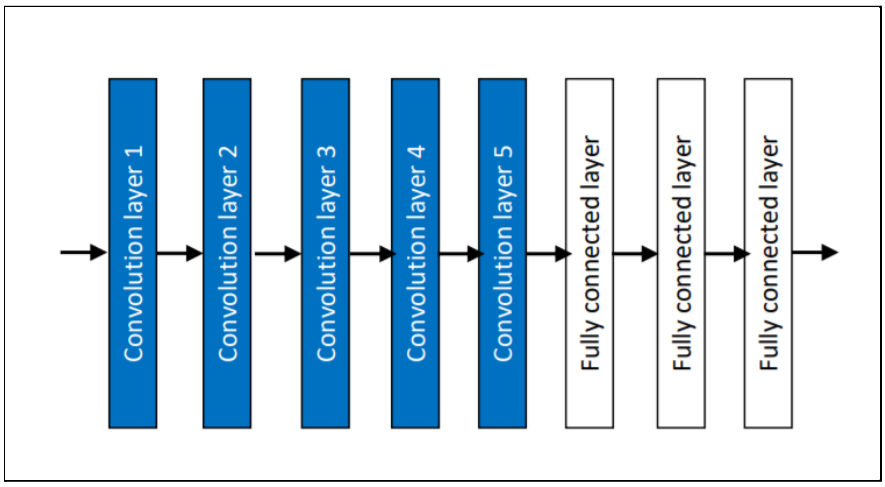
These problems with deep CNNs were resolved through the innovation of the **ResNet**. It essentially shortens the path to the output.

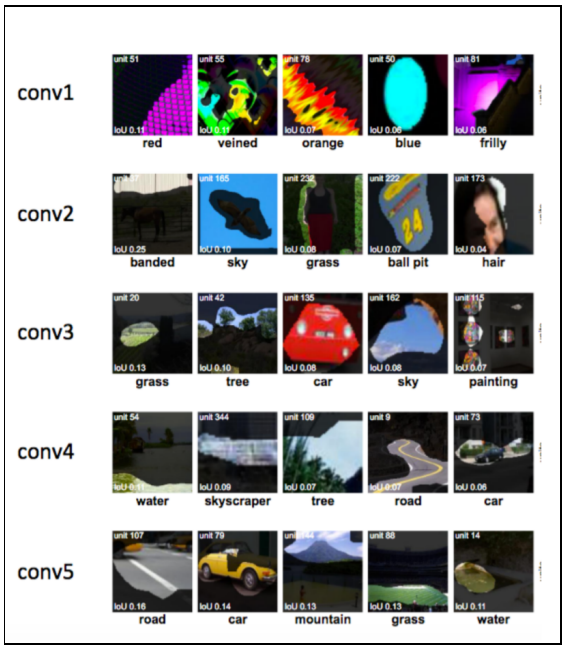
- In ResNets, we have skip connections where after the convolution, we directly add a layer's input itself to the output, thus resulting in skip connections which show improved training for CNNs.
- These skip connections lead to a shorter path between the input and output, which helps us transfer more information during the backpropagation phase.

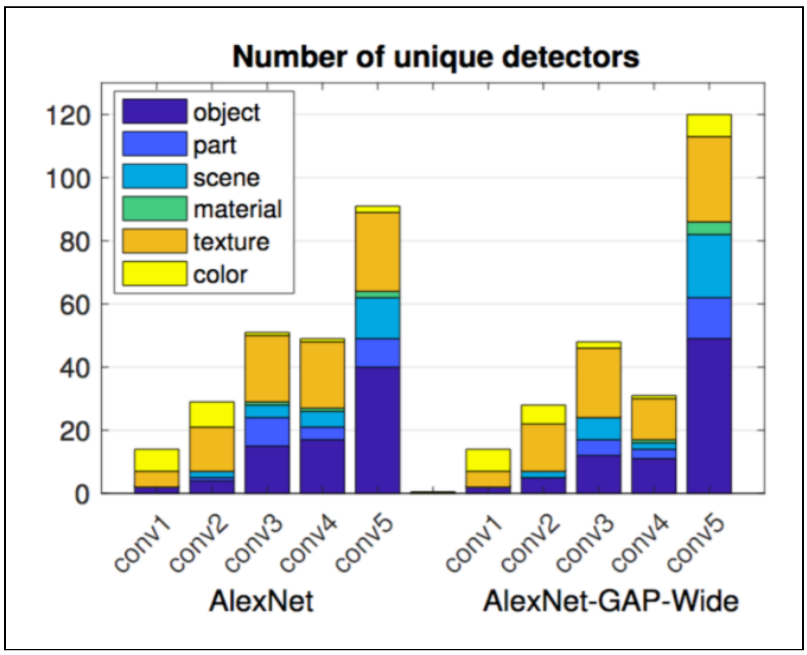


What do neurons in a CNN Learn?

- Neurons in a CNN help in detecting the patterns in data. They follow a hierarchical process of learning.
- The initial layer detects the edges, the next layer combines these edges to detect shapes, then the following layer detects parts and finally, objects are detected.
- In the example below, we use the famous **AlexNet** architecture to try and classify scenes in the images on the right.

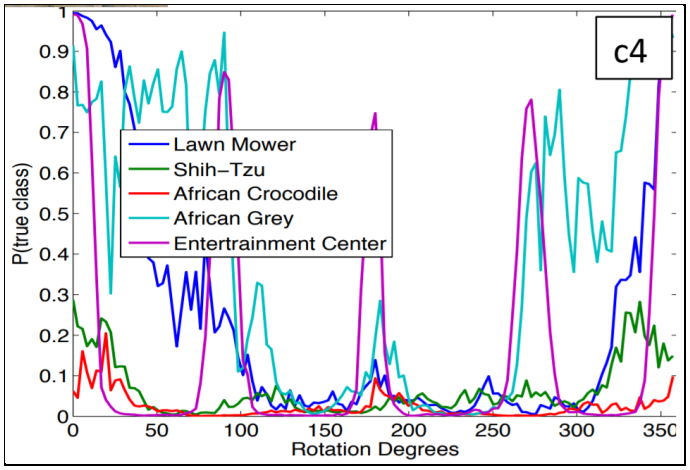


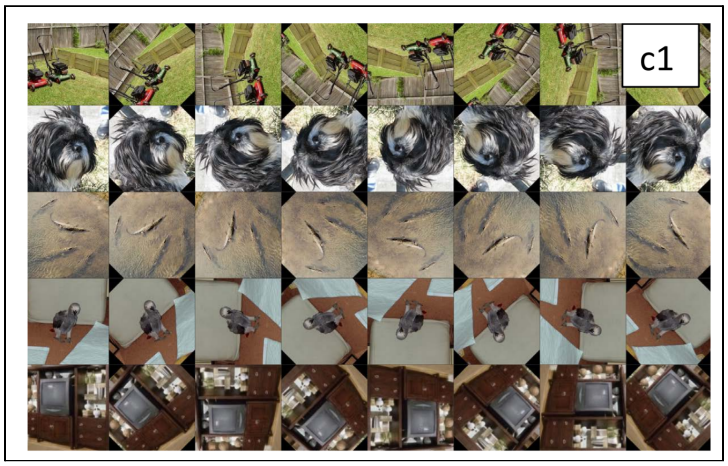




- From the above two images, we observe that the initial layers were able to learn simple features like color and texture, whereas the final layers were able to learn complex objects and scenes.
- This hierarchical learning pattern allows us to obtain a scene detector through which we are able to detect scenes in an image, a huge accomplishment that was not even imaginable a couple of decades ago.

- However, there are also other shortcomings in neural networks - namely they are not robust to **input rotations.** This means that sometimes, the performance of a CNN decays a lot when we feed rotated images to the neural net and try to detect them.



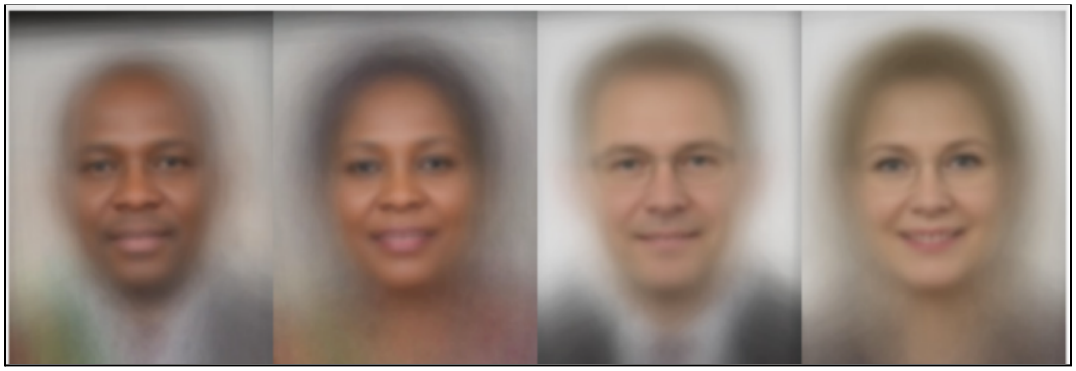


- In the above plot, we can clearly see that as the degree of rotation increases, the accuracy of the model in predicting the output class decreases.
- It is interesting to observe that only the TV's class (Entertainment Center) shows consistent spikes (good performance) around rotations which are multiples of 90 degrees, such as 90, 180, 270 etc. This could be because the shape of the TV is a square, and squares look the same when rotated by multiples of 90 degrees. So perhaps the CNN can detect the TV correctly even after rotation as long as the rotation is of a multiple of 90 degrees.
- Hence, CNNs can sometimes fail when the input is rotated. While they are translationally invariant, they are not rotationally invariant.
- In order to solve this problem, we can provide rotated images as part of the training dataset itself, so that the model understands that even a rotated version of the image still belongs to the same class.

There is also one other thing that we need to keep in mind while using CNNs, and that is the concept of **Bias.**

Bias

- The main concept around bias is that the model's errors can sometimes be very different for different classes and in different situations.
- Let us try to understand this by using an example comparing Gender Classifiers used by Microsoft, Face++ and IBM.



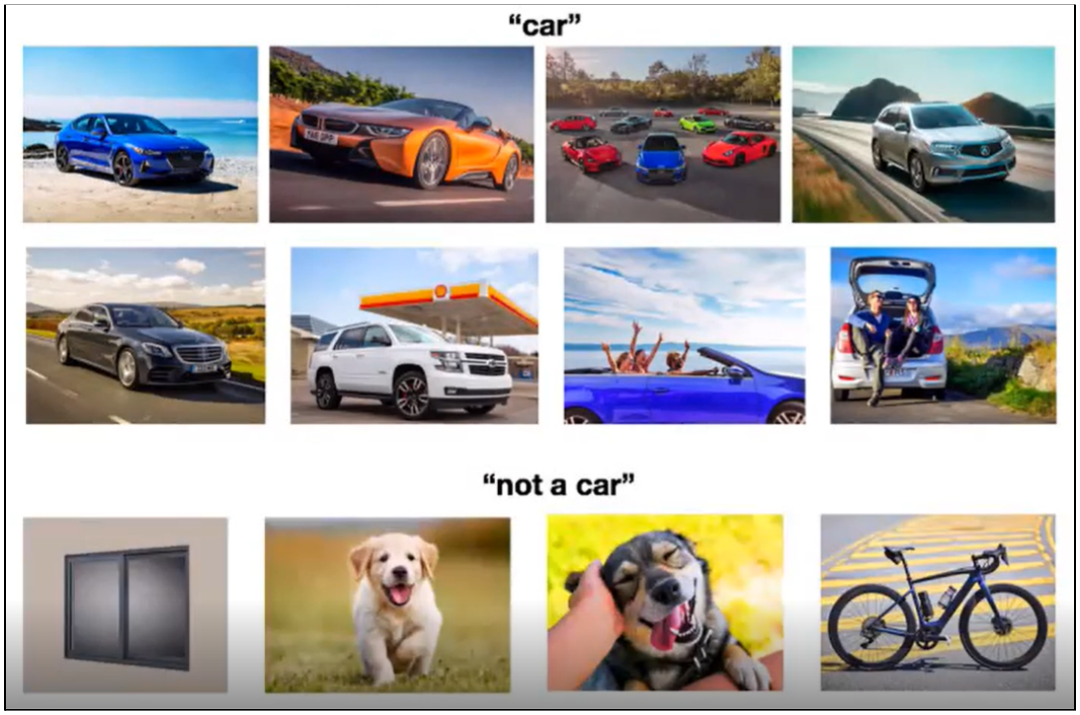
Gender Classifier	Darker Male	Darker Female	Lighter Male	Lighter Female	Largest Gap
Microsoft	94.0%	79.2%	100%	98.3%	20.8%
FACE++	99.3%	65.5%	99.2%	94.0%	33.8%
IBM	88.0%	65.3%	99.7%	92.9%	34.4%

- While the main problem statement is to classify people by gender, we have different subclasses in people's images, such as Darker Male, Darker Female, Lighter Male, and Lighter Female.
- The issue seen above is that the gender classifiers used by these companies worked relatively well on all the other subclasses except for the Darker Female category.
- It was observed that the best performance was in the Lighter Male category, whereas the worst performance was observed on Darker Females.
- This shows us that while we may sometimes get good model accuracies on the whole, there may be some subclasses in the data where our model performs badly.
- This might happen when we have more and well-represented data for lighter males whereas the data may not be that numerous or well-represented for darker females.
- Hence possible reasons for this issue:
    - It may happen due to imbalanced data.
    - It may even happen if the network learns unnecessary features, ignoring the important features which are truly necessary for detection.

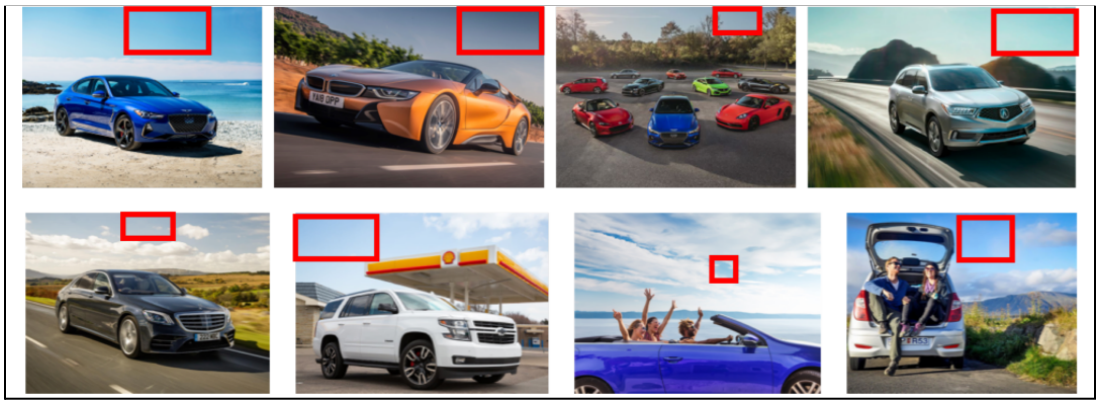
Let us try to learn this using an example to use the correct features for learning.

Are we using the correct features for learning?

- We will try to use a dataset that acts as a classifier for two classes - Car and not a Car.

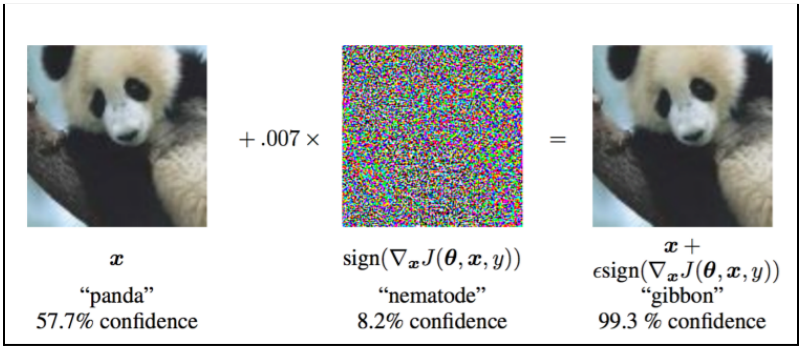


- Our neural network should learn the required features from these images to classify an image as a car or not a car.
- It may happen that our CNN detector does not learn to detect cars in the image but it may learn to detect the sky in the image, since all the images of cars also have a sky in them.



- These are called **shortcut solutions,** as neural networks sometimes fall into the trap of using these shortcut tricks to classify objects or images.

- Similarly, this has been known to happen even in medical data, when there was a case study for detecting Pneumonia from X-rays.
- A CNN model was trained on this data; this model predicted well on our training data but when the model was used on unseen data, it performed very badly. When this problem was further investigated, it was found that the model was still able to make predictions well on X-ray images of patients from the same hospital, but this wasn't the case for patient X-ray images from other hospitals. This means the model has likely learned an unrequired feature from the data by using shortcuts, and that is why it is not able to generalize well to X-ray images of patients from other hospitals.
- Also, it is sometimes observed that when there is some small noise added to the input image, the model does not give us the correct output. For example: when a small noise is added as an input to a panda image, the model predicts the panda as a gibbon.



- The following measures may be followed in order to avoid such problems and make sure the right features are learned:
  - Collecting more data
  - Using data augmentation
  - Pre-training
- We can use larger datasets, or add image augmentation like rotation and cropping to the data to make the model perform better and increase the robustness of the model.
- Examples of data augmentation can be observed by the different image transformations seen below on the same image. All these variations will be fed into and then learned by the model.