**IMPLEMENTATION**

**MODULES:**

* Dataset
* Importing the necessary libraries
* Retrieving the images
* Splitting the dataset
* Building the model
* Apply the model and plot the graphs for accuracy and loss
* Accuracy on test set
* Saving the Trained Model

**MODULES DESCSRIPTION:**

**Dataset:**

In the first module, we developed the system to get the input dataset for the training and testing purpose. We have taken the dataset from brain tumour detection Link — https://www.kaggle.com/navoneel/brain-mri-images-for-brain-tumor-detection

The dataset consists of 3000 brain tumour images

**Importing the necessary libraries:**

We will be using Python language for this. First we will import the necessary libraries such as keras for building the main model, sklearn for splitting the training and test data, PIL for converting the images into array of numbers and other libraries such as pandas, numpy ,matplotlib and tensorflow.

**Retrieving the images:**

We will retrieve the images and their labels. Then resize the images to (224,224) as all images should have same size for recognition. Then convert the images into numpy array.

**Splitting the dataset:**

Split the dataset into train and test. 80% train data and 20% test data.

## Convolutional Neural Networks

The objectives behind the first module of the course 4 are:

* To understand the convolution operation
* To understand the pooling operation
* Remembering the vocabulary used in convolutional neural networks (padding, stride, filter, etc.)
* Building a convolutional neural network for multi-class classification in images

## Computer Vision

Some of the computer vision problems which we will be solving in this article are:

1. Image classification
2. Object detection
3. Neural style transfer

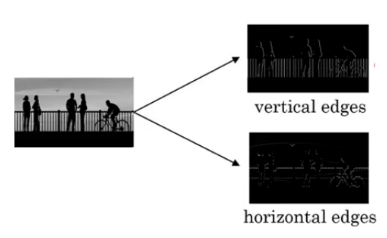
One major problem with computer vision problems is that the input data can get really big. Suppose an image is of the size 68 X 68 X 3. The input feature dimension then becomes 12,288. This will be even bigger if we have larger images (say, of size 720 X 720 X 3). Now, if we pass such a big input to a neural network, the number of parameters will swell up to a HUGE number (depending on the number of hidden layers and hidden units). This will result in more computational and memory requirements – not something most of us can deal with.

Edge Detection Example

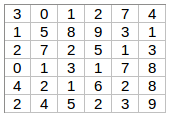
In the previous article, we saw that the early layers of a neural network detect edges from an image. Deeper layers might be able to detect the cause of the objects and even more deeper layers might detect the cause of complete objects (like a person’s face).

In this section, we will focus on how the edges can be detected from an image. Suppose we are given the below image:

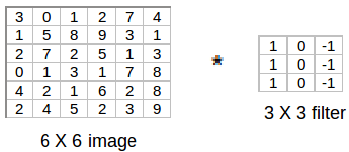


As you can see, there are many vertical and horizontal edges in the image. The first thing to do is to detect these edges:

But how do we detect these edges? To illustrate this, let’s take a 6 X 6 grayscale image (i.e. only one channel):



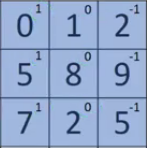
Next, we convolve this 6 X 6 matrix with a 3 X 3 filter:



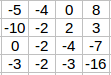
After the convolution, we will get a 4 X 4 image. The first element of the 4 X 4 matrix will be calculated as:



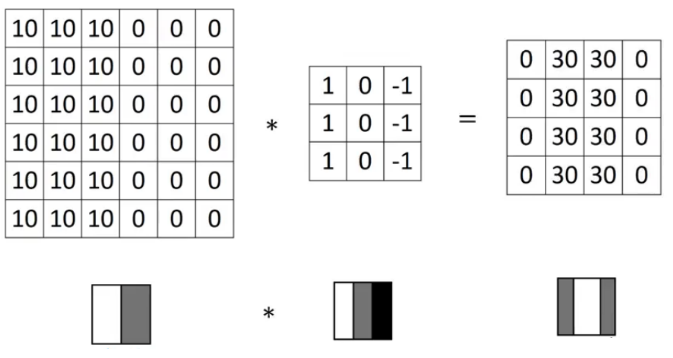
So, we take the first 3 X 3 matrix from the 6 X 6 image and multiply it with the filter. Now, the first element of the 4 X 4 output will be the sum of the element-wise product of these values, i.e. 3\*1 + 0 + 1\*-1 + 1\*1 + 5\*0 + 8\*-1 + 2\*1 + 7\*0 + 2\*-1 = -5. To calculate the second element of the 4 X 4 output, we will shift our filter one step towards the right and again get the sum of the element-wise product:



Similarly, we will convolve over the entire image and get a 4 X 4 output:



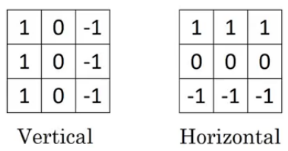
So, convolving a 6 X 6 input with a 3 X 3 filter gave us an output of 4 X 4. Consider one more example:



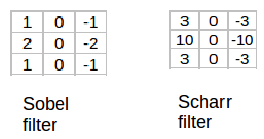
**Note**: Higher pixel values represent the brighter portion of the image and the lower pixel values represent the darker portions. This is how we can detect a vertical edge in an image.

More Edge Detection

The type of filter that we choose helps to detect the vertical or horizontal edges. We can use the following filters to detect different edges:



Some of the commonly used filters are:



The Sobel filter puts a little bit more weight on the central pixels. Instead of using these filters, we can create our own as well and treat them as a parameter which the model will learn using backpropagation.

## Padding

We have seen that convolving an input of 6 X 6 dimension with a 3 X 3 filter results in 4 X 4 output. We can generalize it and say that if the input is n X n and the filter size is f X f, then the output size will be (n-f+1) X (n-f+1):

* **Input:** n X n
* **Filter size:** f X f
* **Output:** (n-f+1) X (n-f+1)

There are primarily two disadvantages here:

1. Every time we apply a convolutional operation, the size of the image shrinks
2. Pixels present in the corner of the image are used only a few number of times during convolution as compared to the central pixels. Hence, we do not focus too much on the corners since that can lead to information loss

To overcome these issues, we can pad the image with an additional border, i.e., we add one pixel all around the edges. This means that the input will be an 8 X 8 matrix (instead of a 6 X 6 matrix). Applying convolution of 3 X 3 on it will result in a 6 X 6 matrix which is the original shape of the image. This is where padding comes to the fore:

* **Input:** n X n
* **Padding:** p
* **Filter size:** f X f
* **Output:** (n+2p-f+1) X (n+2p-f+1)

There are two common choices for padding:

1. **Valid:** It means no padding. If we are using valid padding, the output will be (n-f+1) X (n-f+1)
2. **Same:** Here, we apply padding so that the output size is the same as the input size, i.e.,  
   n+2p-f+1 = n  
   So, p = (f-1)/2

We now know how to use padded convolution. This way we don’t lose a lot of information and the image does not shrink either. Next, we will look at how to implement strided convolutions.

## Strided Convolutions

Suppose we choose a stride of 2. So, while convoluting through the image, we will take two steps – both in the horizontal and vertical directions separately. The dimensions for stride *s* will be:

* **Input:** n X n
* **Padding:** p
* **Stride:** s
* **Filter size:** f X f
* **Output:** [(n+2p-f)/s+1] X [(n+2p-f)/s+1]

Stride helps to reduce the size of the image, a particularly useful feature.

Convolutions Over Volume

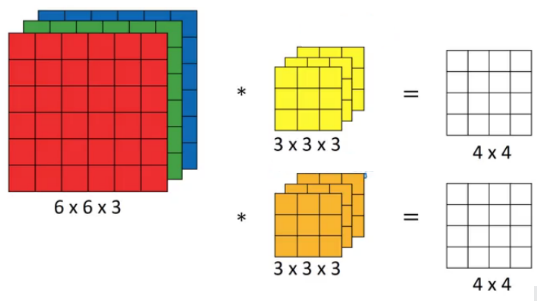
Suppose, instead of a 2-D image, we have a 3-D input image of shape 6 X 6 X 3. How will we apply convolution on this image? We will use a 3 X 3 X 3 filter instead of a 3 X 3 filter. Let’s look at an example:

* **Input:** 6 X 6 X 3
* **Filter:** 3 X 3 X 3

The dimensions above represent the height, width and channels in the input and filter. Keep in mind that the number of channels in the input and filter should be same***.*** This will result in an output of 4 X 4. Let’s understand it visually:

Since there are three channels in the input, the filter will consequently also have three channels. After convolution, the output shape is a 4 X 4 matrix. So, the first element of the output is the sum of the element-wise product of the first 27 values from the input (9 values from each channel) and the 27 values from the filter. After that we convolve over the entire image.

Instead of using just a single filter, we can use multiple filters as well. How do we do that? Let’s say the first filter will detect vertical edges and the second filter will detect horizontal edges from the image. If we use multiple filters, the output dimension will change. So, instead of having a 4 X 4 output as in the above example, we would have a 4 X 4 X 2 output (if we have used 2 filters):



Generalized dimensions can be given as:

* **Input:** n X n X nc
* **Filter:** f X f X nc
* **Padding:** p
* **Stride:** s
* **Output:** [(n+2p-f)/s+1] X [(n+2p-f)/s+1] X nc’

Here, nc is the number of channels in the input and filter, while nc’ is the number of filters.

## One Layer of a Convolutional Network

Once we get an output after convolving over the entire image using a filter, we add a bias term to those outputs and finally apply an activation function to generate activations. *This is one layer of a convolutional network*. Recall that the equation for one forward pass is given by:

z[1] = w[1]\*a[0] + b[1]  
a[1] = g(z[1])

In our case, input (6 X 6 X 3) is a[0]and filters (3 X 3 X 3) are the weights w[1]. These activations from layer 1 act as the input for layer 2, and so on. Clearly, the number of parameters in case of convolutional neural networks is independent of the size of the image. It essentially depends on the filter size. Suppose we have 10 filters, each of shape 3 X 3 X 3. What will be the number of parameters in that layer? Let’s try to solve this:

* Number of parameters for each filter = 3\*3\*3 = 27
* There will be a bias term for each filter, so total parameters per filter = 28
* As there are 10 filters, the total parameters for that layer = 28\*10 = 280

No matter how big the image is, the parameters only depend on the filter size. Awesome, isn’t it? Let’s have a look at the summary of notations for a convolution layer:

* f[l] = filter size
* p[l] = padding
* s[l] = stride
* n[c][l] = number of filters

Let’s combine all the concepts we have learned so far and look at a convolutional network example.

Simple Convolutional Network Example

We’ll take things up a notch now. Let’s look at how a convolution neural network with convolutional and pooling layer works. Suppose we have an input of shape 32 X 32 X 3:

We take an input image (size = 39 X 39 X 3 in our case), convolve it with 10 filters of size 3 X 3, and take the stride as 1 and no padding. This will give us an output of 37 X 37 X 10. We convolve this output further and get an output of 7 X 7 X 40 as shown above. Finally, we take all these numbers (7 X 7 X 40 = 1960), unroll them into a large vector, and pass them to a classifier that will make predictions. This is a microcosm of how a convolutional network works.

There are a number of hyperparameters that we can tweak while building a convolutional network. These include the number of filters, size of filters, stride to be used, padding, etc. We will look at each of these in detail later in this article. Just keep in mind that as we go deeper into the network, the size of the image shrinks whereas the number of channels usually increases.

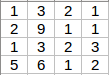
In a convolutional network (ConvNet), there are basically three types of layers:

1. Convolution layer
2. Pooling layer
3. Fully connected layer

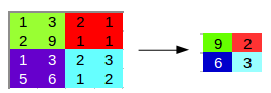
Let’s understand the pooling layer in the next section.

Pooling Layers

Pooling layers are generally used to reduce the size of the inputs and hence speed up the computation. Consider a 4 X 4 matrix as shown below:



Applying max pooling on this matrix will result in a 2 X 2 output:



For every consecutive 2 X 2 block, we take the max number. Here, we have applied a filter of size 2 and a stride of 2. These are the hyperparameters for the pooling layer. Apart from max pooling, we can also apply average pooling where, instead of taking the max of the numbers, we take their average. In summary, the hyperparameters for a pooling layer are:

1. Filter size
2. Stride
3. Max or average pooling

If the input of the pooling layer is nh X nw X nc, then the output will be [{(nh – f) / s + 1} X {(nw – f) / s + 1} X nc].

CNN Example

We’ll take things up a notch now. Let’s look at how a convolution neural network with convolutional and pooling layer works. Suppose we have an input of shape 32 X 32 X 3:

There are a combination of convolution and pooling layers at the beginning, a few fully connected layers at the end and finally a softmax classifier to classify the input into various categories. There are a lot of hyperparameters in this network which we have to specify as well.

Generally, we take the set of hyperparameters which have been used in proven research and they end up doing well. As seen in the above example, the height and width of the input shrinks as we go deeper into the network (from 32 X 32 to 5 X 5) and the number of channels increases (from 3 to 10).

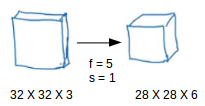
All of these concepts and techniques bring up a very fundamental question – why convolutions? Why not something else?

Why Convolutions?

There are primarily two major advantages of using convolutional layers over using just fully connected layers:

1. Parameter sharing
2. Sparsity of connections

Consider the below example:



If we would have used just the fully connected layer, the number of parameters would be = 32\*32\*3\*28\*28\*6, which is nearly equal to 14 million! Makes no sense, right?

If we see the number of parameters in case of a convolutional layer, it will be = (5\*5 + 1) \* 6 (if there are 6 filters), which is equal to 156. Convolutional layers reduce the number of parameters and speed up the training of the model significantly.

In convolutions, we share the parameters while convolving through the input. The intuition behind this is that a feature detector, which is helpful in one part of the image, is probably also useful in another part of the image. So a single filter is convolved over the entire input and hence the parameters are shared.

The second advantage of convolution is the sparsity of connections. For each layer, each output value depends on a small number of inputs, instead of taking into account all the inputs.

**Building the model:**

For building the we will use sequential model from keras library. Then we will add the layers to make convolutional neural network. In the first 2 Conv2D layers we have used 32 filters and the kernel size is (5,5).

In the MaxPool2D layer we have kept pool size (2,2) which means it will select the maximum value of every 2 x 2 area of the image. By doing this dimensions of the image will reduce by factor of 2. In dropout layer we have kept dropout rate = 0.25 that means 25% of neurons are removed randomly.

We apply these 3 layers again with some change in parameters. Then we apply flatten layer to convert 2-D data to 1-D vector. This layer is followed by dense layer, dropout layer and dense layer again. The last dense layer outputs 2 nodes as the brain tumour or not. This layer uses the softmax activation function which gives probability value and predicts which of the 2 options has the highest probability.

**Apply the model and plot the graphs for accuracy and loss:**

We will compile the model and apply it using fit function. The batch size will be 2. Then we will plot the graphs for accuracy and loss. We got average validation accuracy of 97.6% and average training accuracy of 99.3%.

**Accuracy on test set:**

We got an accuracy of 99.7% on test set

**Saving the Trained Model:**

Once you’re confident enough to take your trained and tested model into the production-ready environment, the first step is to save it into a .h5 or .pkl file using a library like pickle .

Make sure you have pickle installed in your environment.

Next, let’s import the module and dump the model into.pkl file