Demystifying the Mathematics Behind Convolutional Neural Networks (CNNs)

Overview

- Convolutional neural networks (CNNs) are all the rage in the deep learning and computer vision community
- How does this CNN architecture work? We'll explore the math behind the building blocks of a convolutional neural network
- We will also build your own CNN from scratch using NumPy

Introduction

Convolutional neural network (CNN) – almost sounds like an amalgamation of biology, art and mathematics. In a way, that's exactly what it is (and what this article will cover).

CNN-powered deep learning models are now ubiquitous and you'll find them sprinkled into various computer vision applications across the globe. Just like XGBoost and other popular machine learning algorithms, convolutional neural networks came into the public consciousness through a hackathon (the ImageNet competition in 2012).

These neural networks have caught inspiration like fire since then, expanding into various research areas. Here are just a few popular computer vision applications where CNNs are used:

- Facial recognition systems
- Analyzing and parsing through documents
- Smart cities (traffic cameras, for example)
- Recommendation systems, among other use cases

But why does a convolutional neural network work so well? How does it perform better than the traditional ANNs (Artificial neural network)? Why do deep learning experts love it?

To answer these questions, we must understand how a CNN actually works under the hood. In this article, we will go through the mathematics behind a CNN model and we'll then build our own CNN from scratch.

If you prefer a course format where we cover this content in stages, you can enrol in this free course: [Convolutional Neural Networks from Scratch]

Note: If you're new to neural networks, I highly recommend checking out our popular free course:

- *[Introduction to Neural Networks]*

Table of Contents

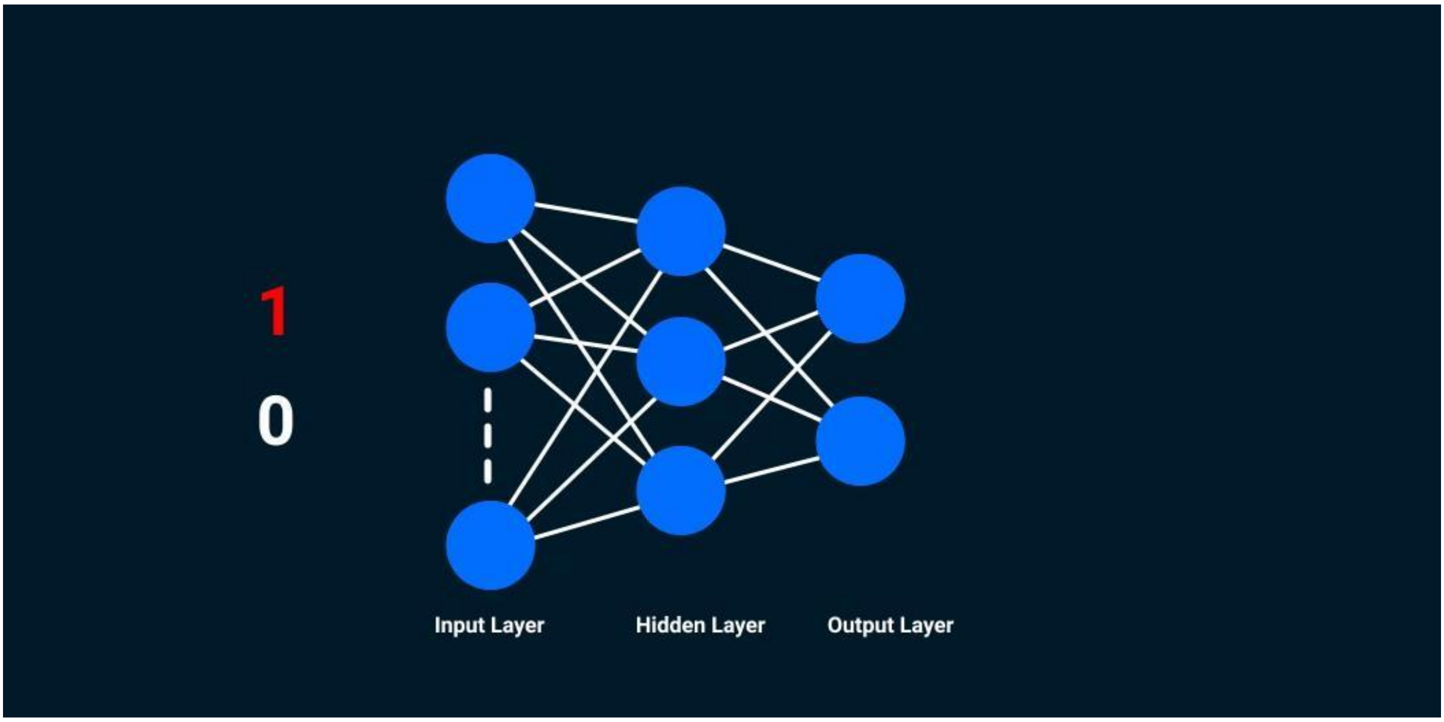
2. Back ward Propagation

Introduction to Neural Networks

Neural Networks are at the core of all [deep learning algorithms](). But before you deep dive into these algorithms, it's important to have a good understanding of the concept of [neural networks]().

These neural networks try to mimic the human brain and its learning process. Like a brain takes the input, processes it and generates some output, so does the neural network.

These three actions – **receiving input, processing information, generating output** – are represented in the form of layers in a neural network – input, hidden and output. Below is a skeleton of what a neural network looks like:



These individual units in the layers are called **neurons**. The complete training process of a neural network involves two steps.

1. Forward Propagation

I mages are fed in to th e in pu t layer in th e form of n u mbers. T h ese n u merical valu es den ote th e in ten sity of pixels in th e image. T h e n eu ron s in th e h idden layers apply a few math ematical operation s on th ese valu es (wh ich we will discu ss later in th is ar ticle).

I n order to perform th ese math ematical operation s, th ere are cer tain parameter valu es th at are ran domly in itialized. Post th ese math ematical operation s at th e h idden layer, th e resu lt is sen t to th e ou tpu t layer wh ich gen erates th e fin al prediction .

2. Backward Propagation

On ce th e ou tpu t is gen erated, th e n ext step is to compare th e ou tpu t with th e actu al valu e. Based on th e fin al ou tpu t, an d h ow close or far th is is from th e actu al valu e (error), th e valu es of th e parameters are u pdated. T h e forward propagation process is repeated u sin g th e u pdated parameter valu es an d n ew ou tpu ts are gen erated.

T h is is th e base of an y n eu ral n etwork algorith m. I n th is ar ticle, we will look at th e forward an d back ward propagation steps for a con volu tion al n eu ral n etwork !

Convolutional Neural Network (CNN) Architecture

C on sider th is – you are ask ed to iden tify objects in two given images. How wou ld you go abou t doin g th at? Typically, you wou ld observe th e image, try to iden tify differen t featu res, sh apes an d edges from th e image. Based on th e in formation you gath er, you wou ld say th at th e object is a dog or a car an d so on .

T h is is precisely wh at th e h idden layers in a C N N do – fin d featu res in th e image. T h e con volu tion al n eu ral n etwork can be brok en down in to two par ts:

- **The conv ol uti on l ay er s**: Extracts featu res from th e in pu t
- **The f ul l y connected (dense) l ay er s**: U ses data from con volu tion layer to gen erate ou tpu t

A s we discu ssed in th e previou s section , th ere are two impor tan t processes in volved in th e train in g of an y n eu ral n etwork :

1. **F or war d P r opagati on:** Receive in pu t data, process th e in formation , an d gen erate ou tpu t
2. **Backwar d P r opagati on:** C alcu late error an d u pdate th e parameters of th e n etwork

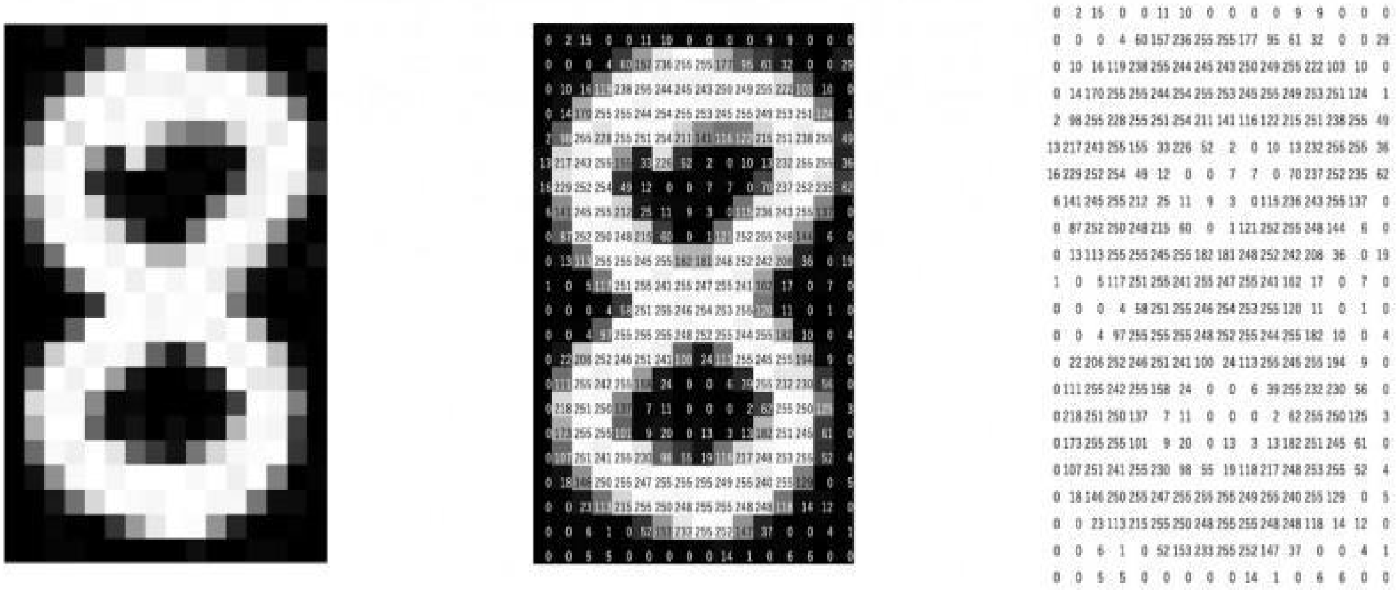
W e will cover both of th ese on e by on e. Let u s star t with th e forward propagation process.

Convolutional Neural Network (CNN): Forward Propagation

Convolution Layer

You know how we look at images and identify the object's shape and edges? A convolutional neural network does this by comparing the pixel values.

Below is an image of the number 8 and the pixel values for this image. Take a look at the image closely. You would notice that there is a significant difference between the pixel values around the edges of the number. Hence, a simple way to identify the edges is to compare the neighboring pixel value.



Do we need to traverse pixel by pixel and compare these values? No! To capture this information, the image is convolved with a filter (also known as a 'kernel').

Convolution is often represented mathematically with an asterisk * sign. If we have an input image represented as X and a filter represented with *f*, then the expression would be:

```
Z = X * f
```

Note: To learn how filters capture information about the edges, you can go through this article:

- [Beginner-Friendly Techniques to Extract Features from Image Data]

Let us understand the process of convolution using a simple example. Consider that we have an image of size 3 x 3 and a filter of size 2 x 2:

$$\begin{bmatrix} 1 & 7 & 2 \\ 11 & 1 & 23 \\ 2 & 2 & 2 \end{bmatrix} \qquad \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

T h e filter goes th rou gh th e patch es of images, performs an elemen t-wise mu ltiplication , an d th e valu es are su mmed u p:

```
(1x1 + 7x1 + 11x0 + 1x1) = 9 (7x1 + 2x1 + 1x0 + 23x1)  = 32 (11x1  + 1x1 + 2x0 + 2x1) = 14 (1x1 + 23x1 + 2x0 +
2x1) = 26
```

$$\begin{bmatrix} \boxed{\begin{matrix} 1 & 7 \\ 11 & 1 \end{matrix}} & \begin{matrix} 2 \\ 23 \end{matrix} \\ \begin{matrix} 2 & 2 \end{matrix} & 2 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 9 & \\ & \end{bmatrix}$$

Look at th at closely – you ’ll n otice th at th e filter is con siderin g a small por tion of th e image at a time. W e can also imagin e th is as a sin gle image brok en down in to smaller patch es, each of wh ich is con volved with th e filter.

$$\begin{bmatrix} \boxed{\begin{matrix} 1 & 7 \\ 11 & 1 \end{matrix}} & \begin{matrix} 2 \\ 23 \end{matrix} \\ \begin{matrix} 2 & 2 \end{matrix} & 2 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

I n th e above example, we h ad an in pu t of sh ape (3, 3) an d a filter of sh ape (2, 2). Sin ce th e dimen sion s of image an d filter are very small, it’s easy to in terpret th at th e sh ape of th e ou tpu t matrix is (2, 2). Bu t h ow wou ld we fin d th e sh ape of an ou tpu t for more complex in pu ts or filter dimen sion s? T h ere is a simple formu la to do so:

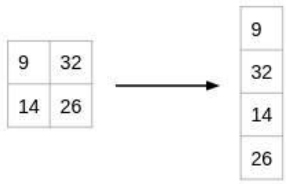
```
Dimension  of image = (n, n) Dimension  of filter  = (f,f) Dimension  of output  will be ((n-f+1)  , (n-f+1))
```

You sh ou ld h ave a good u n derstan din g of h ow a con volu tion al layer work s at th is poin t. Let u s move to th e n ext par t of th e C N N arch itectu re.

Fully Connected Layer

So far, th e con volu tion layer h as extracted some valu able featu res from th e data. T h ese featu res are sen t to th e fu lly con n ected layer th at gen erates th e fin al resu lts. T h e fu lly con n ected layer in a C NN is n oth in g bu t th e tradition al n eu ral n etwork !

T h e ou tpu t from th e con volu tion layer was a 2D matrix. I deally, we wou ld wan t each row to represen t a sin gle in pu t image.   I n fact, th e fu lly con n ected   layer can on ly work   with   1D data. Hen ce, th e valu es gen erated from th e previou s operation are first con ver ted in to a 1D format.

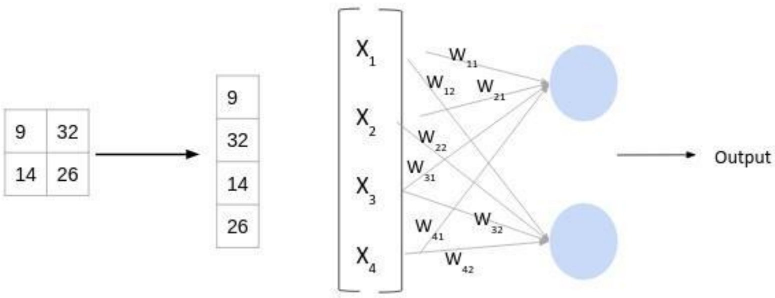


On ce th e data is con ver ted in to a 1D array, it is sen t to th e fu lly con n ected layer. A ll of th ese in dividu al valu es are treated as separate featu res th at represen t th e image. T h e fu lly con n ected layer performs two operation s on th e in comin g data – **a l i near tr ansf or mati on and a non-l i near tr ansf or mati on.**

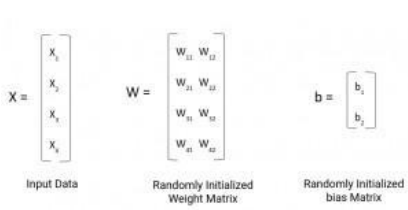
W e first perform a lin ear tran sformation on th is data. T h e equ ation for lin ear tran sformation is:

$Z = W^T.X + b$

Here, X is th e in pu t, W is weigh t, an d b (called bias) is a con stan t. Note th at th e W in th is case will be a matrix of (ran domly in itialized) n u mbers. C an you gu ess wh at wou ld be th e size of th is matrix?



C on siderin g th e size of th e matrix is (m, n) – m will be equ al to th e n u mber of featu res or in pu ts for th is layer. Sin ce we h ave 4 featu res from th e con volu tion layer, m h ere wou ld be 4. T h e valu e of n will depen d on th e n u mber of n eu ron s in th e layer. F or in stan ce, if we h ave two n eu ron s, th en th e sh ape of weigh t matrix will be (4, 2):



Havin g defin ed th e weigh t an d bias matrix, let u s pu t th ese in th e equ ation for lin ear tran sformation :

$$Z = W^T . X + b$$

$$Z = \begin{bmatrix} W_{11} & W_{21} & W_{31} & W_{41} \\ W_{12} & W_{22} & W_{32} & W_{42} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

$$Z_{2x2} = \begin{bmatrix} W_{11}X_1 + W_{21}X_2 + W_{31}X_3 + W_{41}X_4 \\ W_{12}X_1 + W_{22}X_2 + W_{32}X_3 + W_{42}X_4 \end{bmatrix}$$

Now, th ere is on e fin al step in th e forward propagation process – th e n on -lin ear tran sformation s. Let u s u n derstan d th e con cept of n on -lin ear tran sformation an d it's role in th e forward propagation process.

Non-Linear transformation

T h e lin ear tran sformation alon e can n ot captu re complex relation sh ips. T h u s, we in trodu ce an addition al compon en t in th e n etwork wh ich adds n on -lin earity to th e data. T h is n ew compon en t in th e arch itectu re is called th e **acti v ati on f uncti on**.

T h ere are a n u mber of activation fu n ction s th at you can u se – h ere is th e complete list:

- F u n d a m e n t a l s o f D e e p L e a r n i n g – A c t i v a t i o n F u n c t i o n s a n d W h e n t o U s e T h e m ?

T h ese activation fu n ction s are added at each layer in th e n eu ral n etwork . T h e activation fu n ction to be u sed will depen d on th e type of problem you are solvin g.

W e will be work in g on a bin ary classification problem an d will u se th e Sigmoid activation fu n ction . Let's qu ick ly look at th e math ematical expression for th is:

```
f(x) = 1/(1+e^-x)
```

T h e ran ge of a Sigmoid fu n ction is between 0 an d 1. T h is mean s th at for an y in pu t valu e, th e resu lt wou ld always be in th e ran ge (0, 1). A Sigmoid fu n ction is majorly u sed for bin ary classification problems an d we will u se th is for both con volu tion an d fu lly-con n ected layers.

Let's qu ick ly su mmarize wh at we've covered so far.

Forward Propagation Summary

Step 1: Load th e in pu t images in a variable (say X)

Step 2: Defin e (ran domly in itialize) a filter matrix. I mages are con volved with th e filter

```
Z1 = X * f
```

Havin g defin ed th e weigh t an d bias matrix, let u s pu t th ese in th e equ ation for lin ear tran sformation :

Step 3: A pply th e Sigmoid activation fu n ction on th e resu lt

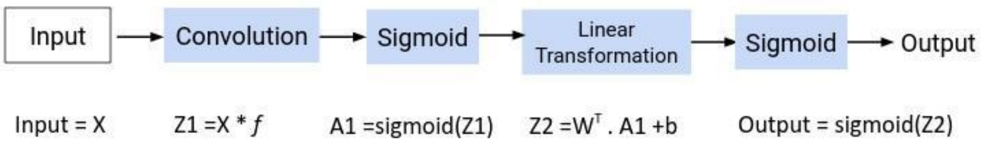
```
A = sigmoid(Z1)
```

Step 4: Define (randomly initialize) weight and bias matrix. Apply linear transformation on the values

```
Z2 = W^T.A + b
```

Step 5: Apply the Sigmoid function on the data. This will be the final output

```
O = sigmoid(Z2)
```



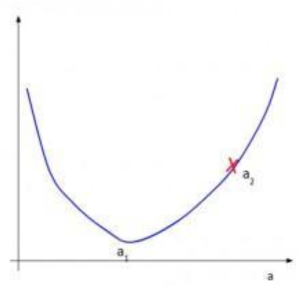
Now the question is – how are the values in the filter decided? The CNN model treats these values as parameters, which are randomly initialized and learned during the training process. We will answer this in the next section.

Convolutional Neural Network (CNN): Backward Propagation

During the forward propagation process, we randomly initialized the weights, biases and filters. These values are treated as parameters from the convolutional neural network algorithm. **In the backward propagation process, the model tries to update the parameters such that the overall predictions are more accurate.**

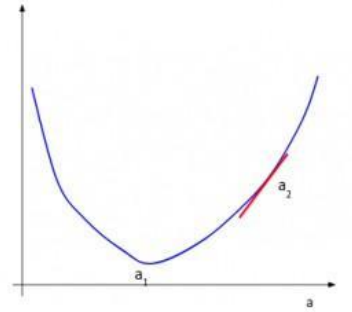
For updating these parameters, we use the [gradient descent technique](). Let us understand the concept of gradient descent with a simple example.

Consider that following in the curve for our loss function where we have a parameter a:



During the random initialization of the parameter, we get the value of a as a_2. It is clear from the picture that the minimum value of loss is at a_1 and not a_2. The gradient descent technique tries to find this value of parameter (a) at which the loss is minimum.

We understand that we need to update the value a_2 and bring it closer to a_1. To decide the direction of movement, i.e. whether to increase or decrease the value of the parameter, we calculate the gradient or slope at the current point.



Based on the value of the gradient, we can determine the updated parameter values. When the slope is negative, the value of the parameter will be increased, and when the slope is positive, the value of the parameter should be decreased by a small amount.

Here is a generic equation for updating the parameter values:

```
new_parameter = old_parameter - (learning_rate * gradient_of_parameter)
```

The learning rate is a constant that controls the amount of change being made to the old value. The slope or the gradient determine the direction of the new values, that is, should the values be increased or decreased. So, we need to find the gradients, that is, change in error with respect to the parameters in order to update the parameter values.

If you want to read about the gradient descent technique in detail, you can go through the below article:

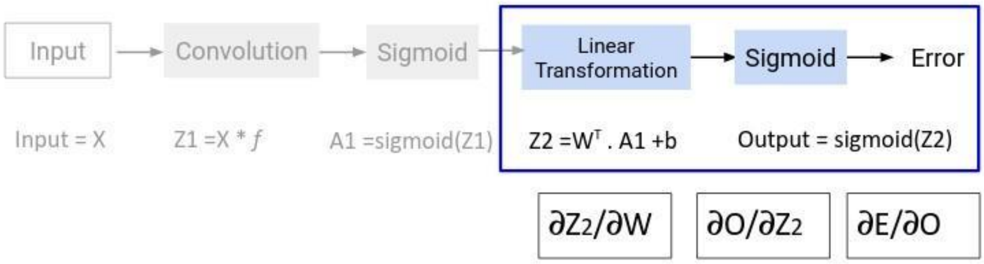
- [Introduction to Gradient Descent Algorithm]

We know that we have three parameters in a CNN model – weights, biases and filters. Let us calculate the gradients for these parameters one by one.

Backward Propagation: Fully Connected Layer

As discussed previously, the fully connected layer has two parameters – weight matrix and bias matrix. Let us start by calculating the change in error with respect to weights – ∂E/ ∂W.

Since the error is not directly dependent on the weight matrix, we will use the concept of chain rule to find this value. The computation graph shown below will help us define ∂E/ ∂W :



Backward Propagation (Fully Connected layer)

$\partial E/\partial W = \partial E/\partial O \ .\ \partial O/\partial Z2.\ \ \partial z/\partial W$

We will find the values of these derivatives separately.

1. Change in error with respect to output

Suppose the actual values for the data are denoted as y' and the predicted output is represented as O. Then the error would be given by this equation :

$E = (y' - O)^2/2$

If we differentiate the error with respect to the output, we will get the following equation :

$\partial E/\partial O = -(y'-O)$

2. Change in output with respect to Z_2 (linear transformation output)

To find the derivative of output O with respect to Z_2, we must first define O in terms of Z_2. If you look at the computation graph from the forward propagation section above, you would see that the output is simply the sigmoid of Z_2. Thus, $\partial O/\partial Z_2$ is effectively the derivative of Sigmoid. Recall the equation for the Sigmoid function :

$f(x) = 1/(1+e^{-x})$

The derivative of this function comes out to be:

$f'(x) = (1+e^{-x})^{-1}[1-(1+e^{-x})^{-1}]$ $f'(x) = sigmoid(x)(1-sigmoid(x))$ $\partial O/\partial Z2 = (O)(1-O)$

You can read abou t th e complete derivation of th e Sigmoid fu n ction h e r e .

3. Change in Z_2 with respect to W (Weights)

The value Z_2 is the result of the linear transformation process. Here is the equation of Z_2 in terms of weights:

```
Z2 = W^T.A1 + b
```

On differentiating Z_2 with respect to W, we will get the value A_1 itself:

```
∂Z2/∂W = A1
```

Now that we have the individual derivations, we can use the chain rule to find the change in error with respect to weights:

```
∂E/∂W = ∂E/∂O . ∂O/∂Z2. ∂Z2/∂W ∂E/∂W = -(y'-o) . sigmoid'. A1
```

The shape of ∂E/ ∂W will be the same as the weight matrix W. We can update the values in the weight matrix using the following equation :

```
W_new = W_old - lr*∂E/∂W
```

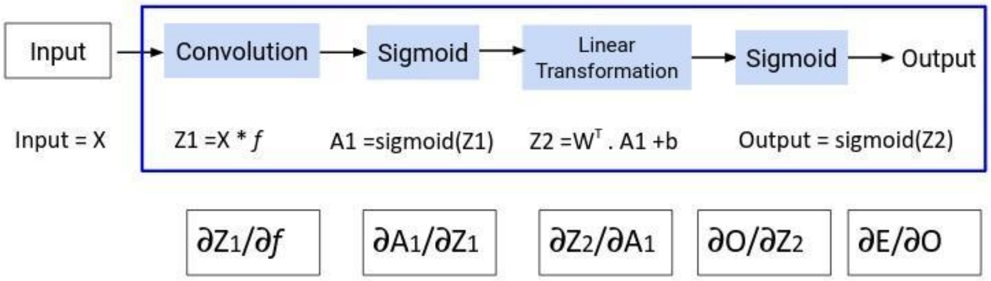
Updating the bias matrix follows the same procedure. Try to solve that yourself and share the final equations in the comments section below!

Backward Propagation: Convolution Layer

For the convolution layer, we had the filter matrix as our parameter. During the forward propagation process, we randomly initialized the filter matrix. We will now update these values using the following equation :

```
new_parameter = old_parameter - (learning_rate * gradient_of_parameter)
```

To update the filter matrix, we need to find the gradient of the parameter – dE/ df. Here is the computation graph for backward propagation :



Backward Propagation (Convolution layer)

From the above graph we can define the derivative $\partial E/\partial f$ as:

```
∂E/∂f = ∂E/∂O.∂O/∂Z2.∂Z2/∂A1 .∂A1/∂Z1.∂Z1/∂f
```

We have already determined the values for $\partial E/\partial O$ and $\partial O/\partial Z2$. Let us find the values for the remaining derivatives.

1. Change in Z_2 with respect to A_1

To find the value for $\partial Z_2/\partial A_1$, we need to have the equation for Z_2 in terms of A_1:

$$Z2 = W^T.A1 + b$$

On differentiating the above equation with respect to A_1, we get W^T as the result:

$$\partial Z2/\partial A1 = W^T$$

2. Change in A_1 with respect to Z_1

The next value that we need to determine is $\partial A_1/\partial Z_1$. Have a look at the equation of A_1

```
A1 = sigmoid(Z1)
```

This is simply the Sigmoid function. The derivative of Sigmoid would be:

```
∂A1/∂Z1 = (A1)(1-A1)
```

3. Change in Z_1 with respect to filter *f*

Finally, we need the value for $\partial Z_1/\partial f$. Here's the equation for Z_1

```
Z1 = X * f
```

Differentiating Z with respect to X will simply give us X:

```
∂Z1/∂f = X
```

Now that we have all the required values, let's find the overall change in error with respect to the filter:

```
∂E/∂f   = ∂E/∂O.∂O/∂Z2.∂Z2/∂A1 .∂A1/∂Z1  * ∂Z1/∂f
```

Notice that in the equation above, the value $(\partial E/\partial O.\ \partial O/\partial Z2.\ \partial Z2/\partial A1\ .\ \partial A1/\partial Z)$ is convolved with $\partial Z1/\partial f$ instead of using a simple dot product. Why? The main reason is that during forward propagation, we perform a convolution operation for the images and filters.

This is repeated in the backward propagation process. Once we have the value for $\partial E/\partial f$, we will use this value to update the original filter value:

```
f = f - lr*(∂E/∂f)
```

This completes the backpropagation section for convolutional neural networks. It's now time to code!

CNN from Scratch using NumPy

Excited to get your hands dirty and design a convolutional neural network from scratch? The wait is over!

We will start by loading the required libraries and dataset. Here, **we will be using the MNIST dataset which is present within the *keras.datasets* library.**

For the purpose of this tutorial, we have selected only the first 200 images from the dataset. Here is the distribution of classes for the first 200 images:

```python
# importing required libraries
import numpy as np
import pandas as pd
from tqdm import tqdm
from keras.datasets import mnist

# loading dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# selecting a subset of data (200 images)
x_train = x_train[:200]
y = y_train[:200]

X = x_train.T
X = X/255

y.resize((200,1))
y = y.T

#checking value
pd.Series(y[0]).value_counts()
```

view raw

loading_mnist.py hosted with ❤ by GitHub

```
1 26 9 23 7 21 4 21 3 21 0 21 2 20 6 19 8 15 5 13 dtype:  int64
```

As you can see, we have ten classes here – 0 to 9. **This is a multi-class classification problem**. For now, we will start with building a simple CNN model for a binary classification problem:

```python
# converting into binary classification
for i in range(y.shape[1]):
```

```
    if y[0][i] >4:
        y[0][i] = 1
```

```
    else:
        y[0][i] = 0

#checking value counts
pd.Series(y[0]).value_counts()
```

view raw

binary_class.py hosted with ❤ by GitHub

```
0 109 1 91 dtype: int64
```

We will now initialize the filters for the convolution operation :

```
# initializing filter
f=np.random.uniform(size=(3,5,5))
f = f.T

print('Filter 1', '\n', f[:,:,0], '\n')
print('Filter 2', '\n', f[:,:,1], '\n')
print('Filter 3', '\n', f[:,:,2], '\n')
```

view raw

filter_initialization.py hosted with ❤ by GitHub

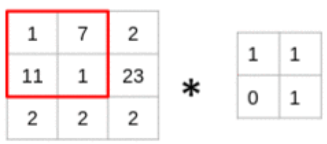
Let us quickly check the shape of the loaded images, target variable and the filter matrix:

```
X.shape, y.shape, f.shape
```

```
((28, 28, 200), (1, 200), (5, 5, 3))
```

We have 200 images of dimensions (28, 28) each. For each of these 200 images, we have one class specified and hence the shape of y is (1, 200). Finally, we defined 3 filters, each of dimensions (5, 5).

The next step is to prepare the data for the convolution operation. As we discussed in the forward propagation section, the image-filter convolution can be treated as a single image being divided into multiple patches:



For every single image in the data, we will create smaller patches of the same dimension as the filter matrix, which is (5, 5). Here is the code to perform this task :

```
# Generating patches from images
new_image = []
```

```
# for number  of images
```

```python
for k in range(X.shape[2]):
    # sliding  in horizontal  direction
    for i in range(X.shape[0]-f.shape[0]+1):
        # sliding  in vertical  direction
        for j in range(X.shape[1]-f.shape[1]+1):
            new_image.append(X[:,:,k][i:i+f.shape[0],j:j+f.shape[1]])

# resizing  the generated  patches  as per number  of images
new_image  = np.array(new_image)
new_image.resize((X.shape[2],int(new_image.shape[0]/X.shape[2]),new_image.shape[1],new_image.shape[2]))
new_image.shape
```

view raw

create_patches.py hosted with ❤ by GitHub

```
(200, 576, 5, 5)
```

We have everything we need for the forward propagation process of the convolution layer. Moving on to the next section for forward propagation, we need to initialize the weight matrix for the fully connected layer. The size of the weight matrix will be (m, n) – where m is the number of features as input and n will be the number of neurons in the layer.

What would be the number of features? We know that we can determine the shape of the output image using this formula:

```
((n-f+1)  , (n-f+1))
```

Since the fully connected layer only takes 1D input, we will flatten this matrix. This means the number of features or input values will be:

```
((n-f+1)  x (n-f+1)  x num_of_filter)
```

Let us initialize the weight matrix:

```python
# number  of features  in data set
s_row  = X.shape[0]  - f.shape[0]  + 1
s_col  = X.shape[1]  - f.shape[1]  + 1
num_filter  = f.shape[2]

inputlayer_neurons = (s_row)*(s_col)*(num_filter)
output_neurons = 1

# initializing weight
wo=np.random.uniform(size=(inputlayer_neurons,output_neurons))
```

view raw

weight.py hosted with ❤ by GitHub

Finally, we will write the code for the activation function which we will be using for the convolution neural network architecture.

Now, we have already converted the original problem into a binary classification problem. Hence, we will be using Sigmoid as our activation function. We have already discussed the mathematical equation for Sigmoid and its derivative. Here is the python code for the same:

```python
# defining  the Sigmoid  Function
def sigmoid  (x):
    return  1/(1 + np.exp(-x))

# derivative  of Sigmoid  Function
def derivatives_sigmoid(x):
    return  x * (1 - x)
```

view raw

sigmoid.py hosted with ❤ by GitHub

Great! We have all the elements we need for the forward propagation process. Let us put these code blocks together.

First, we perform the convolution operation on the patches created. After the convolution, the results are stored in the form of a list, which is converted into an array of dimension (200, 3, 576). Here

- 200 is the number of images
- 576 is the number of patches created, and
- 3 is the number of filters we used

After the convolution operation, we apply the Sigmoid activation function:

```
# generating  output  of convolution layer
filter_output = []
# for each image
for i in range(len(new_image)):
     # apply  each filter
     for k in range(f.shape[2]):
          # do element  wise multiplication
          for j in range(new_image.shape[1]):
               filter_output.append((new_image[i][j]*f[:,:,k]).sum())

filter_output = np.resize(np.array(filter_output), (len(new_image),f.shape[2],new_image.shape[1]))

# applying  activation  over convolution output
filter_output_sigmoid = sigmoid(filter_output)

filter_output.shape, filter_output_sigmoid.shape
```

view raw

convolution.py hosted with ❤ by GitHub

```
((200,  3, 576), (200, 3, 576))
```

After the convolution layer, we have the fully connected layer. We know that the fully connected layer will only have 1D inputs. So, we first flatten the results from the previous layer using the *reshape* function. Then, we apply the linear transformation and activation function on this data:

```
# generating  input  for fully  connected  layer
filter_output_sigmoid = filter_output_sigmoid.reshape((filter_output_sigmoid.shape[0],filter_output_sigmoid.shape[1]*filter_outp

filter_output_sigmoid = filter_output_sigmoid.T

# Linear  trasnformation for fully  Connected  Layer
output_layer_input= np.dot(wo.T,filter_output_sigmoid)
output_layer_input = (output_layer_input - np.average(output_layer_input))/np.std(output_layer_input)

# activation  function
output  = sigmoid(output_layer_input)
```

view raw

linear.py hosted with ❤ by GitHub

It's time to start the code for backward propagation. Let's define the individual derivatives for backward propagation of the fully connected layer. Here are the equations we need:

$E = (y' - O)^2/2$ $\partial E/\partial O = -(y'-O)$ $\partial O/\partial Z2 = (O)(1-O)$ $\partial Z2/\partial W = A1$

Let us code this in Python:

```
#Error
error  = np.square(y-output)/2

```

```
#Error  w.r.t  Output  (Gradient)
error_wrt_output = -(y-output)
```

```

#Error w.r.t sigmoid transformation (output_layer_input)
output_wrt_output_layer_input=output*(1-output)

#Error w.r.t weight
output_wrt_w=filter_output_sigmoid
```


We have the individual derivatives from the previous code block. We can now find the overall change in error w.r.t. weight using the chain rule. Finally, we will use this gradient value to update the original weight matrix:

```
W_new = W_old - lr*∂E/∂W
```

```
#delta change in w for fully connected layer
delta_error_fcp = np.dot(output_wrt_w,(error_wrt_output * output_wrt_output_layer_input).T)

wo = wo - lr*delta_error_fcp
```


So far we have covered back propagation for the fully connected layer. This covers updating the weight matrix. Next, we will look at the derivatives for back propagation for the convolutional layer and update the filters:

$\partial E/\partial f = \partial E/\partial O.\partial O/\partial Z2.\partial Z2/\partial A1 .\partial A1/\partial Z1 * \partial Z1/\partial f$ $\partial E/\partial O = -(y'-O)$ $\partial O/\partial Z2 = (O)(1-O)$ $\partial Z2/\partial A1 = W^T$ $\partial A1/\partial Z1= A1(1-A1)$
$\partial Z1/\partial f = X$

We will code the first four equations in Python and calculate the derivative using the *np.dot* function. Post that we need to perform a convolution operation using $\partial Z_1 / \partial f$:

```
#Error w.r.t sigmoid output
output_layer_input_wrt_filter_output_sigmoid = wo.T

#Error w.r.t sigmoid transformation
filter_output_sigmoid_wrt_filter_output = filter_output_sigmoid * (1-filter_output_sigmoid)

# cvalculating derivatives for backprop convolution
error_wrt_filter_output = np.dot(output_layer_input_wrt_filter_output_sigmoid.T,error_wrt_output*output_wrt_output_layer_input)
error_wrt_filter_output = np.average(error_wrt_filter_output, axis=1)
error_wrt_filter_output = np.resize(error_wrt_filter_output,(X.shape[0]-f.shape[0]+1,X.shape[1]-f.shape[1]+1, f.shape[2]))
```



```
filter_update = []
for i in range(f.shape[2]):
    for j in range(f.shape[0]):
        for k in range(f.shape[1]):
            temp = 0
            spos_row = j
            spos_col = k
            epos_row = spos_row + s_row
            epos_col = spos_col + s_col
            for l in range(X.shape[2]):
                temp = temp + (X[spos_row:epos_row,spos_col:epos_col,l]*error_wrt_filter_output[:,:,i]).sum()
            filter_update.append(temp/X.shape[2])

filter_update_array = np.array(filter_update)
filter_update_array = np.resize(filter_update_array,(f.shape[2],f.shape[0],f.shape[1]))
```


We now have the gradient value. Let us use it to update the error:

```python
for i in range(f.shape[2]):
    f[:,:,i]  = f[:,:,i]  - lr*filter_update_array[i]
```

view raw

filter_update.py hosted with ❤ by GitHub

End Notes

Take a deep breath – that was a lot of learning in one tutorial! Convolutional neural networks can appear to be slightly complex when you're starting out but once you get the hang of how they work, you'll feel ultra confident in yourself.

I had a lot of fun writing about what goes on under the hood of these CNN models we see everywhere these days. We covered the mathematics behind these CNN models and learned how to implement them from scratch, using just the NumPy library.

I want you to explore other concepts related to CNN, such as the padding and pooling techniques. Implement them in the code as well and share your ideas in the comments section below.

You should also try your hand at the below hackathons to practice what you've learned:

- [Practice Problem: Identify the Apparels]
- [Practice Problem: Identify the Digits]
